Investigations into Playing Chess Endgames using Reinforcement Learning.

by

Richard Peter Dazeley (BCOMP)

A Dissertation submitted to the School of Computing in Partial fulfillment of the requirements for the degree of

Bachelor of Computing with Honours

University of Tasmania
(October, 2001)

Declaration

This thesis contains no material, which has been accepted for the award of any other degree or diploma in any tertiary institution, and that, to my knowledge and belief, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Signed_____

Abstract

Research in computer game playing has relied primarily on brute force searching approaches rather than any formal AI method. However, these methods may not be able to exceed human ability, as they need human expert knowledge to perform as well as they do. One recently popularized field of research known as reinforcement learning has shown good prospects in overcoming these limitations when applied to non-deterministic games.

This thesis investigated whether the TD(_) algorithm, one method of reinforcement learning, using standard back-propagation neural networks for function generalization, could successfully learn a deterministic game such as chess. The aim is to determine if an agent using no external knowledge can learn to defeat a random player consistently.

The results of this thesis suggests that, even though the agents faced a highly information sparse environment, an agent using a well selected view of the state information was still able to learn to not only to differentiate between various terminating board positions but also to improve its play against a random player. This shows that the reinforcement learning techniques are quite capable of learning behaviour in large deterministic environments without needing any external knowledge.

Acknowledgements

There are a number of people I would like to acknowledge for assisting and supporting me throughout my period of Honours study, without which this work would not be what it is today.

My supervisors Peter Vamplew and Ray Williams, for their patience, encouragement and insight.

The technical staff, David Herbert and Luke Fletcher for repeatedly recovering lost data from my rogue agents and for their general support.

The rest of the academic staff for their support and encouragement.

The Honours students for their companionship.

Edward for the eight months of incessant whistling, out of tune singing, and load clapping. A headache I apparently needed.

My mother and grandmother for their support.

Uisce and Akalia for their love and affection.

I especially would like to thank Anitra for her support during some exceptionally difficult times.

To
Uisce and **Akalia-Shen**

Contents

List of Figures

List of Equations

List of Tables

CHAPTER 1 Introduction

The chessboard is the world, the pieces are the phenomena of the Universe, the rules of the game are what we call the laws of Nature and the player on the other side is hidden from us.

Thomas Huxley

Since the advent of computers people have been seeking ways to make these machines think and learn. There have been numerous methods applied to this problem with varying degrees of success. Many of the more successful methods used have relied primarily on existing human knowledge such as expert systems and neural networks. The problem of relying on human knowledge is that these systems are unable to learn beyond the knowledge of their teacher as they cannot investigate problems themselves. Methods that do not require expert knowledge such as genetic algorithms have had some success in learning solutions to specific problems but do not learn from their interaction with an external environment.

Reinforcement learning is a relatively new field of research that has become popular over the last decade as a means of creating agents capable of learning behaviour, with little or no outside knowledge, through interaction with an external environment. Basically, the agent works by observing the current state of the environment and deciding on an action, if any, to be taken. Then the agent continues over a period of time to observe the environment in order to judge whether its position has improved or not in relation to its eventual goal. It must make this judgment itself without assistance from an expert. At no time is it informed whether an action is correct or not or what the correct action would have been.

Previous work in the field has usually applied the techniques to problems with small state spaces, where all the states can be represented individually in memory. Alternatively, when larger state spaces are used, the problems are usually non-deterministic, because they have been found to learn well due to their ability to measure probabilities in uncertain environments. When applied to deterministic problems, knowledge is usually introduced in order to counter this lack of uncertainty and produce a more viable product.

There has been little work published looking at reinforcement learning using external knowledge when applied to deterministic problems with large state spaces. This thesis is interested in investigating this area by testing whether reinforcement learning is capable of deriving any useful information in an extremely sparse environment using a purist approach of not introducing knowledge. In order to achieve this objective the board game chess (1.2) was selected as the problem domain.

Chess is recognized as being one of the most complicated board games, with more than 10^{16} possible states in just the first 10 moves. Clearly this easily exceeds the capability of even the largest mainframe computer to store all the possible states individually, forcing some form of state generalization to be employed. The deterministic nature of chess also means that agents trying to learn the game cannot rely on any probability variations. Interestingly, it is recognized that there is no 'perfect knowledge' regarding how to play the game. Studies have shown that even grandmasters not only play sub-optimally; they also make mistakes even in simplified endgame situations (Smith, 1999). Finally, chess has the added advantage of having complete state information, which is rarely found in real world problems.

However, due to the time limitations inherit in an Honours project and because the depth of the game tree for chess was potentially too deep to be successful when using a pure approach, it was decided that the scope of chess as a problem domain was too broad. Therefore, the problem was narrowed to encompass a selected chess endgame (1.2.4). Endgames significantly reduce the potential depth of the game tree but do not reduce the state space so far that generalization is no longer required, and therefore maintains the primary objective. Furthermore, endgames are commonly "… agreed to be intellectually the deepest part of chess." (Hayes and Levy, p64, 1976) and thus the core nature of the game is not lost. Finally, the use of endgames has the added advantage of reducing the amount of state information required for input into the agent, thereby, increasing the number of options for board representation.

1.1 Thesis Hypothesis

Specifically, the aim of this thesis is to investigate whether an agent employing reinforcement-learning techniques, using standard backpropagation neural networks for function generalisation, is capable of learning to improve its play in chess endgames. To achieve this aim a number of agents will need to be implemented, each using varying views of the state information or different parameter settings. These agents will be trained through self-play, thereby ensuring no external knowledge is introduced into the system through the agents' environment. Furthermore, the only information the agents will receive is whether they have won, drawn or lost each training game; leaving all interpretation of board position, move selection and strategies up to the agents themselves.

Obviously, due to this extremely information-sparse environment the agents are not expected to play competitive chess. Instead, they will simply be tested for any small evidence of learning to show that the method used can learn in such environments. This will be achieved by first testing how they perform in evaluating the difference between particular board positions. Secondly, the agents will play a number of games against an opponent playing random moves. Originally, before any training has occurred, the agents will basically be moving pieces randomly, therefore, if they improve their play after training then they have learnt from their interactions with the environment.

The central aim of this thesis describes the use of a standard backpropagation neural network for function generalisation. This is required because there are too many states to represent in memory individually. Therefore, we must use some device to group similar states together and give them the same value. While any method could be used, one of the best methods available for performing this generalisation is a neural network. It is important to note that the neural network is simply a tool used by the reinforcement-learning agent and not the agent itself.

1.2 The Game of Chess

Chess is the oldest and, with over half a billion players, one of the most played occidental board games in the world. It has maintained its popularity and has been extensively documented and studied over many centuries. The great German Romantic writer Johann Wolfgang von Goethe called chess "the touchstone of the intellect" (anon, 2001). This section will present a brief history and describe the basic rules of the game to give the reader an understanding of the problem domain being studied in this project. Should the reader wish to find out more about the rules of chess it is recommended they visit the official Fédération Internationale des Échecs site (FIDE, 1999).

1.2.1 History of Chess

The origin of chess is unclear but it is thought to have originated in China or India around the 6th century. It is believed to have evolved from a group of games related to a game called Chaturanga (anon, 2001). Chaturanga is Sanskrit (an ancient Indo-Aryan language) for the four arms (divisions) of an Indian army: elephants, cavalry, chariots, and infantry, which were the precursors to the four military based chess pieces. The game spread through Persia where it was called shatranj (the Arabic form of the word) and on through to Sicily with the spread of Islam. It reached central Europe with the invasion of the Moors into Spain and reached Russia through various trade routes. By the 11th century the game was well known throughout Europe (Op cit).

The modern game of chess emerged in southern Europe toward the end of the 15th century. The new form of the game had some new rules such as castling, the two-square pawn advance and the 'en passant' rule. Also, many pieces were changed in both name and powers available such as the fers counsellor, a weak piece in shatranj, became the Queen, the strongest piece in chess. This new game became popular all over Europe and has had no major alterations since (anon, 2001).

1.2.2 Game Overview

Chess is played between two opponents who take turns to move their pieces on an 8-by-8 grid of alternating light and dark squares, called a chessboard. Each player starts with sixteen pieces, either white or black. The objective of the game is to place the opposing king under attack in such a way that the opponent does not have a legal move that prevents their king from being captured in the following move. The player successfully achieving this goal is said to have 'checkmated' the opponent and won the game.

Additionally, a game can be drawn in a number of circumstances. Firstly, if a player has no legal moves but their king is not under attack then the game is stalemated. Secondly, if the players agree that a result cannot be achieved they can declare the game a draw. Also a draw can be declared if the same position is repeated three times. Finally, if there is a sequence of fifty moves without either a pawn being moved or a piece being captured then it is a draw.

Figure 1.1 shows the standard piece setup for the game of chess with each player's pieces facing each other across the intervening territory.

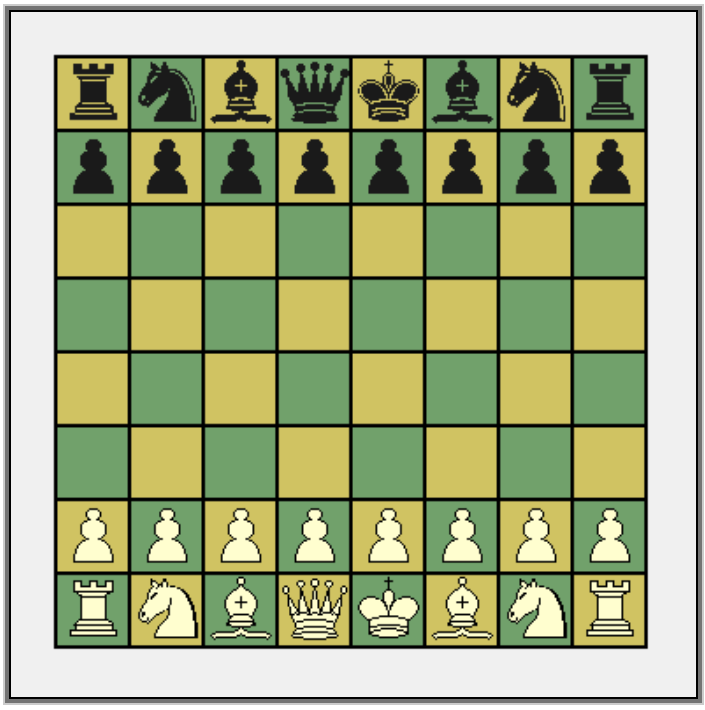


Figure 1.1: Standard board setup for the game of chess.

1.2.3 Piece Movement

There are six different types of pieces making up the initial sixteen pieces allocated to each player. Each type of piece moves in a different way. The variation in type of movement promotes strategic difficulty within the game. Figure 1.2 shows how the king, queen, bishop, knight and rook move. With the exception of the knight, all the pieces slide along the shown paths to a new position and if they encounter a piece along the way then they must either stop before reaching it or capture the piece if its owned by the opposition. The knight jumps from its current position to the new location and is not affected by any intervening pieces. However, it cannot move to a square occupied by one of its own pieces.

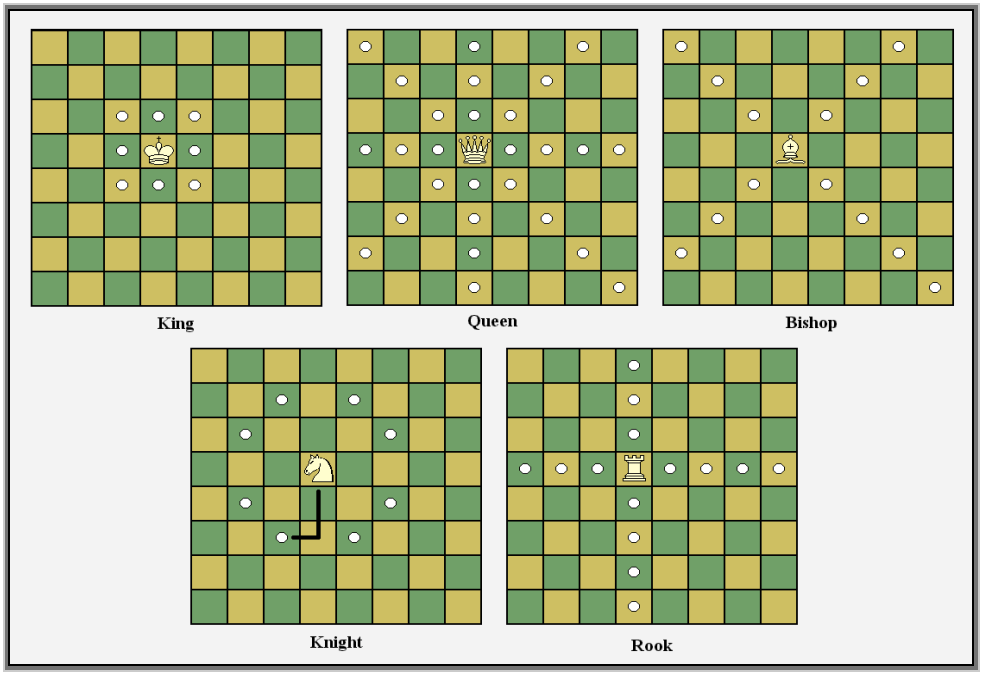


Figure 1.2: Movement of chess pieces.

The pawn can behave differently in various situations. Its basic movement is a forward movement of one square, which it can perform at any time providing the square is not occupied by any piece. A pawn can only capture an opposition piece by moving in a forward-diagonal direction one square at a time. This diagonal move, however, can only be performed when capturing an opponent's piece. The pawn can also advance two squares forward when moving for the first time. Nevertheless, when the double advance move is selected it potentially opens the pawn up to being captured with a special move called 'en passant'.

1.2.3.1 En Passant

The en passant move allows a player to take a pawn that has just advanced two squares with their own pawn providing it would have been able to take the advancing pawn with the standard capture move had it only advanced a single move forward. Figure 1.3 shows this graphically. Here it can be seen that the white pawn has advanced two squares from its starting position. Because the black pawn is adjacent to where the white pawn finished it can capture it as if the white pawn had simply advanced a single move.

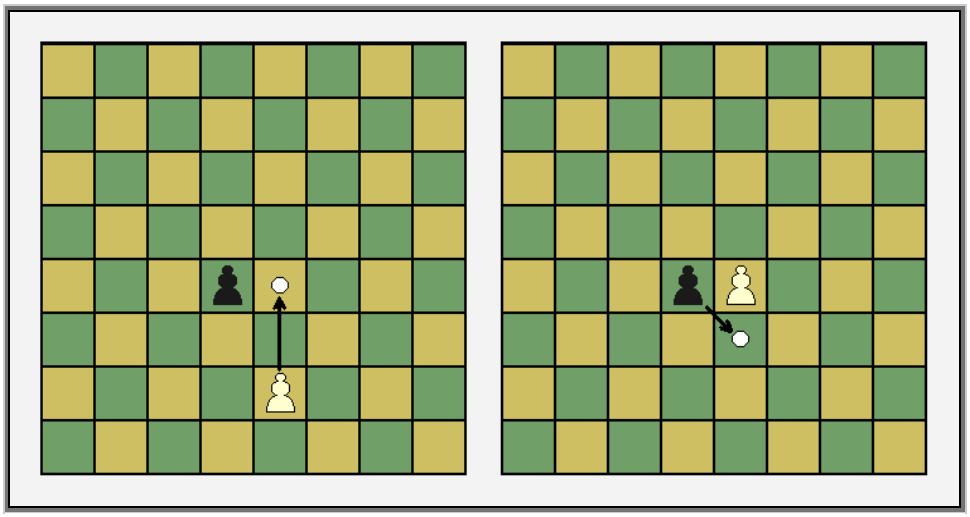


Figure 1.3: Example of en passant move.

1.2.3.2 Pawn Promotion

There is one additional task that can be performed by a Pawn, called ‘promotion’. As the Pawn can only move in a forward direction, the potential exists for it to eventually reach the far edge of the board. When this occurs the piece is promoted to a queen, bishop, knight or rook. The piece it is being promoted to, does not have to be a piece that has already been captured, therefore, it is possible to have more than one queen on the board at a time.

1.2.3.3 Castling

There is also one other special pair of moves called ‘castling’. This move actually allows the player to move two moves simultaneously. Firstly, the king is moved two squares horizontally towards the rook participating in the castling move. Secondly, the rook jumps over the king and is placed in the square next to the king. However, before the move can occur three

requirements must be met. Firstly, neither the king nor the rook it is going to castle with can have moved yet and secondly, the squares in between the king and rook must be cleared of pieces. Finally, none of the squares between the two pieces and the squares occupied by the king and rook themselves can be currently under attack. Figure 1.4 illustrates this set of moves.

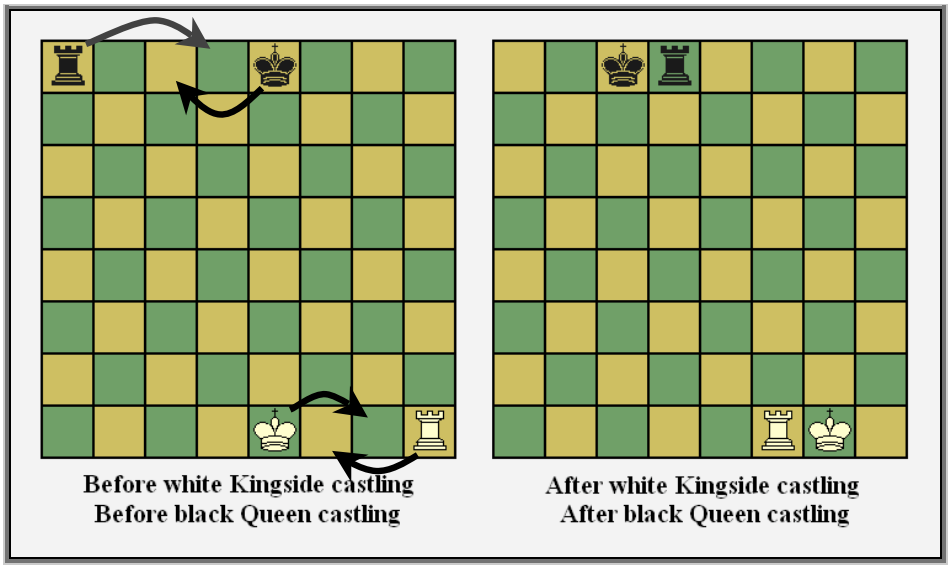


Figure 1.4: Example of Kingside and Queenside castling moves.

1.2.4 Chess Endgames

Generally, a game of chess is regarded as containing three phases of game play: the opening game, the middle game and the endgame and each requires a different set of strategies. In the opening phase each player attempts to gain good board positions for their pieces. During the middle stage they try to initiate piece-swapping sequences, where each player takes a piece from the other, which give them some piece advantage by taking a better piece than is lost. Eventually, most of the pieces have been removed usually leaving only a few pieces for each player. The Endgame is where each player attempts to use those last few remaining pieces to place their opponent into checkmate and win the game. This can be a very difficult stage to play well, with numerous books and examples having been made available.

1.3 Thesis Organisation

The first four chapters present all the introductory and background information required, in reinforcement learning, neural networks and the traditional brute force game algorithm. The next two chapters describe the methods used in this project to implement the intelligent agents and the results gathered. The final chapter discusses the results, highlighting successes, failures, limitations of the algorithm and possible extensions to this work.

- **Chapter 2: Neural Networks**

  This chapter introduces the basics in neural network theory. It describes the mathematical basis and implementation of both single and multiple layer perceptron models. It does not describe any of the more advanced aspects of neural network theory as they are not relevant to this project.

- **Chapter 3: Reinforcement Learning**

  This chapter represents the core of the work being studied in this thesis. It describes in detail the history and development of the major theories behind reinforcement learning. It presents both the mathematics and algorithms used, illustrating how they are applied. Finally, it covers some advanced techniques and describes how these are integrated with neural networks when the problem domain is large.

- **Chapter 4: Computers Playing Games**

  This chapter describes the traditional game algorithm used over the last 50 years, including some of the more subtle alterations that have been used to improve their performance. It also presents a brief overview of a number of case studies of game applications that have had success or been relevant to this field of research.

- **Chapter 5: Methodology**

  This chapter outlines the experiment that has been performed in this thesis. It describes the program design and algorithm used. Finally, it describes in detail how the individual agents were implemented, what view of the state space was used and what variations and parameter changes were applied to each agent to alter their performance.

- **Chapter 6: Results**

  This chapter identifies what data was collected and presents a number of graphs showing which agents performed well. It also discuses a number of the individual results found.

- **Chapter 7: Discussion and Conclusion**

  The final chapter discusses the results found in chapter 6 further and identifies the issues that resulted in various successes and failures. It also highlights what alterations could be made to the methodology used in this thesis along with what changes could be made to the algorithm when applied to a problem such as chess to gain a better performance.

CHAPTER 2 Neural Networks

In the opening a master should play like a book, in the middle game like a magician, in the ending like a machine.

Spielmann

As previously mentioned, reinforcement learning requires a device for storing state values and when the problem domain contains a large state space there must be a means of generalizing these state values. This process of generalization is particularly useful when applied to the game of chess. Chess, like many board games, can have many states that may appear different but are essentially very similar or even identical. For instance, a king that is trapped against the left side of the board by the opposing queen is essentially in the same predicament as if it were trapped against the top, bottom or right sides. If these similar positions could be grouped together so that they are regarded as being essentially the same, then they could be given a single value. This would reduce the memory requirement and increase the speed of learning, as only values for groups of states, rather than individual states, would need to be found.

The practice of grouping similar states is essentially a form of pattern recognition. This process works by taking particular features of the input data and classifying them accordingly. One of the best tools available for performing pattern recognition is a neural network. A reinforcement-learning agent using a neural network for state generalization could, therefore, learn not only state values but also the grouping of those states.

This chapter will first provide a brief history of artificial neural networks (ANNs), followed by an overview of the biology of the human brain to provide a basis for the ANN methods discussed. The following two sections will then describe the basic artificial neuron and the more recently developed multilayer perceptron along with the associated learning rules. Then in the following chapter the integration of the reinforcement learning agent and its neural network will be discussed.

2.1 History of Artificial Neural Networks

McCulloch and Pitts developed the first formal model of an elementary computing neuron in 1943. The model contained all the necessary elements to perform logic operations. While this model was never implemented, as the vacuum tubes of the era were too bulky, it did form the basis for future development in the field (Zurada, 1992).

The first learning rule, proposed by Donald Hebb in 1949, was to update the connections of a neuron and is now referred to as the *Hebbian learning rule*. This was developed further into a neuron-like element called a *perceptron* by Frank Rosenblatt in 1958. Also at this time, the first neurocomputers were built, which were trainable machines capable of learning to classify certain patterns through the modification of connections to the threshold elements (Zurada, 1992).

In the early 1960s a device referred to as the ADALINE (for ADAptive LINEar combiner), and its later extension called MADALINE (for Many ADALINEs) were developed. These devices used a powerful new learning rule, called the *Widrow-Hoff learning rule* developed by Bernard Widrow and Marcian Hoff, and were used in applications including pattern recognition, weather forecasting and adaptive controls (Zurada, 1992). However, at about this time a book by Marvin Minsky and Seymour Papert (1969) revealed the inability of the perceptron model to learn linearly inseparable problems like the *XOR* problem (Beale and Jackson, 1990).

This weakness in the model brought an end to nearly all work in the field until the mid-1980s when Rumelhart and McClelland introduced a new multilayer perceptron and a new learning rule. The introduction of the new multilayer perceptron has revitalised the field, providing an explosion of interest. Numerous applications have been developed such as an EEG spike detector and an autonomous driver (Zurada, 1992).

2.2 The Biological Building Blocks of the Human Brain

The human brain is one of the most studied and yet least understood elements of the universe. We have little understanding of what our mind is and what makes us think. One thing that we do have a rudimentary understanding of though, is how it operates at a low level. The human brain consists of approximately ten thousand million (10^{10}) elementary nerve cells called *neurons*. Each neuron is connected to ten thousand (10^4) other neurons in a very complex biological neural network (Beale and Jackson, 1990).

The neuron is the basic processing unit of the brain and consists of three main components: the cell body or *soma*, the *axon*, and the *dendrites*, as illustrated in Figure 2.1. There are two main types of neuron. The first performs local processing, called *interneuron* cells, which may instead of having an axon simple produce output via the dendrites. The second type called output cells can either connect different regions of the brain to each other, connect the brain to muscle or connect from sensory organs into the brain (Zurada, 1992).

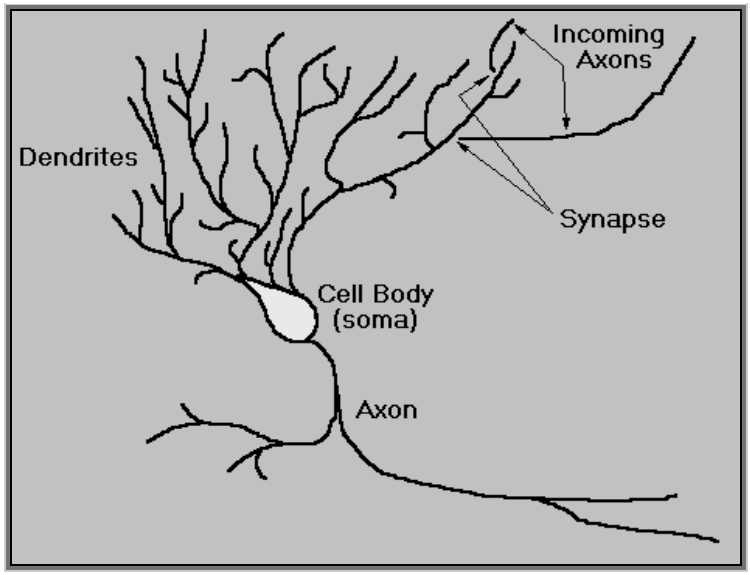


Figure 2.1: Schematic diagram of a single neuron (Zurada, 1992).

The soma is the main body of the neuron and performs the processing function. It gathers its inputs, from the many branching dendrites that form a dendritic tree by repeatedly forking into finer structures, connecting the neuron to many others. Dendrites receive information from neighbouring neurons via their axons. Axons are also long fibres extending from the soma that act as transmission lines. The

axon of one neuron is 'connected' to another's dendrite via a *synapse*. There is no direct physical link in the synaptic connection; rather, there is a temporary chemical one, which when activated either excites or inhibits the dendrite (Zurada, 1992).

The neuron is able to respond to the total of its inputs aggregated within a short period of time called the *period of latent summation*. The cell will be said to 'fire' if the total potential gathered on the dendrites, within the specified time period, exceeds some specified level, or *threshold*. It then generates an output signal, which either inhibits or excites further neurons (Zurada, 1992).

2.3 The Basic Artificial Neuron

The model developed originally was designed to capture the basic features of the neuron and was called the *perceptron*, as proposed by McCulloch and Pitts in 1943. The perceptron adds a set of inputs, that can be on, 1, or off, 0, and if they exceed some value called the *threshold*, denoted θ, then the neuron is said to *fire*, producing an output of 1, otherwise it outputs 0. Each of the inputs can also be altered in intensity by multiplying its value by the weight associated with that input (Beale and Jackson, 1990). Equation 2.1 expresses this mathematically, where w_i is the weight on the i^{th} input and x_i is the i^{th} input, either 0 or 1.

$$\sum_{i=0}^{n} w_i x_i$$

Equation 2.1: Summation of inputs for simple perceptron.

Figure 2.2 (a) shows graphically, how the above type of thresholding is achieved. An alternative method is to instead subtract the threshold value from the weighted-sum and activate the neuron instead when the value is greater than zero, as shown in Figure 2.2 (b). This method allows us to remove the threshold from the body of the neuron and instead have an additional input that is always on. This input will also have a weight, which is the equivalent to $-\theta$. This method is called biasing a neuron and allows our neuron to also adjust the threshold value the same as a weight. The value $-\theta$ is therefore known as the *bias* or *offset* (Beale and Jackson, 1990).

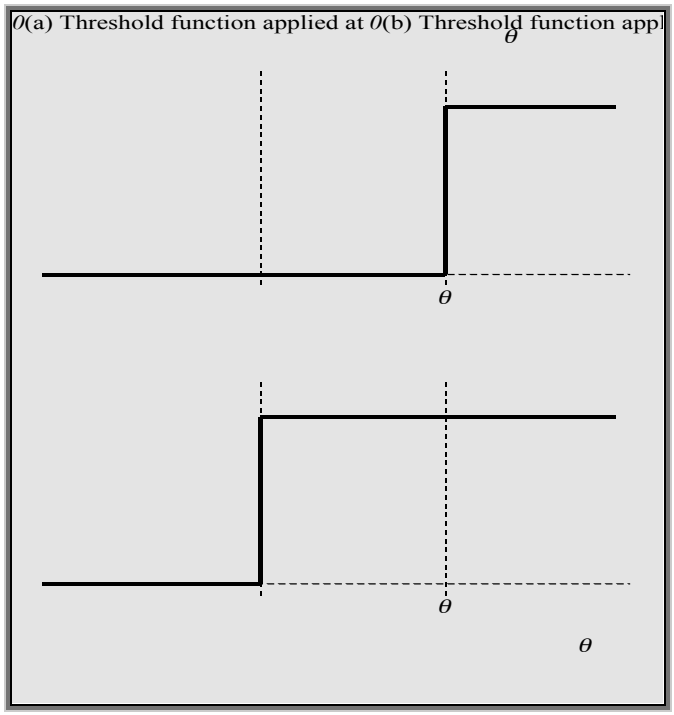


Figure 2.2: The thresholding function (Beale and Jackson, 1990).

Thus, we can define the perceptron formally as in Equation 2.2, where f_h is a step function, or *Heaviside* function, producing either 1 or 0, and y is the neuron's output.

$$y = f_h\left[\sum_{i=0}^{n} w_i x_i - \theta\right],$$

where

$$f_h(x) = 1 \quad x > 0$$
$$f_h(x) = 0 \quad x <= 0$$

Equation 2.2: Formal definition of a perceptron using a bias.

Finally the basic perceptron as just described is shown graphically in Figure 2.3. This diagram shows the inputs x_0 to x_n and the associated weights. The first input, x_0, and weight, w_0, provide the thresholding value and y is the output.

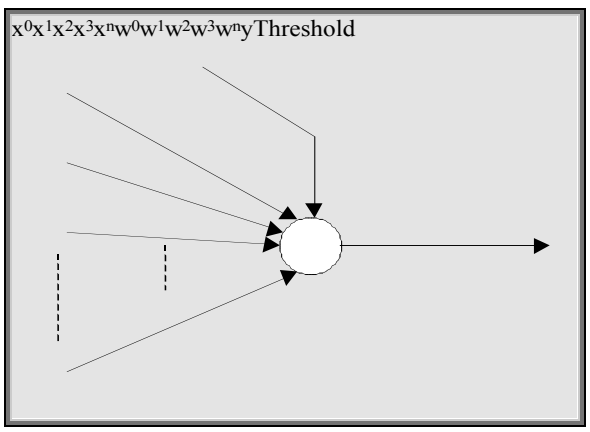


Figure 2.3: Diagram of basic Perceptron using bias.

2.3.1 Learning using the Perceptron

While connecting these neurons together and manually selecting weights for the connections would create a network capable of producing some desired output, it does not achieve anything that conventional programming techniques cannot do already. Their true power comes from their ability to adjust the weights of the connections themselves, using a *learning rule* and thereby, derive their own solution to a problem (Beale and Jackson, 1990).

In biological systems, it can be observed that many people and animals learn when correct behaviour is encouraged and incorrect behaviour discouraged. Essentially, we can train our neural networks in the same manner. When it produces an incorrect answer we reduce the chances of that response recurring and do nothing when a correct output is given (Beale and Jackson, 1990). This is achieved by first allocating each weight with an initial value, usually random, and then presenting the first set of inputs. If an incorrect response is given, for instance a 1 instead of a 0, then the weighted sum is too high and must be reduced, and vice-versa if we get a 0 when we expected a 1 then the weighted sum is too small and must be increased.

This basic algorithm is called *Hebbian learning*, after Donald Hebb, and is given in algorithmic form in Figure 2.4. It may be observed that in this algorithm the weights are not changed if the correct response is given or if the input for that weight was not active. Thus, weights that did not contribute to the incorrect response are not altered (Beale and Jackson, 1990).

1. Initialise weights and threshold

Define $w_i(t)$,where $0 \leq i \leq n$ to be the weight from input i at time t
and θ to be the threshold value in the output node.
Set w_0 to be $-\theta$, the bias, and x_0 to be always 1.
Set $w_i(0)$ to be small random values.

2. Present input and desired output.

Present input $x_0, x_1, x_2, \ldots, x_n$ and the desired output $d(t)$.

3. Calculate actual output using

$$y(t) = f_h\left[\sum_{i=0}^{n} w_i(t)x_i(t)\right]$$

4. Adapt weights:

if correct	$w_i(t+1) = w_i(t)$
if output 0, should be 1 (class A)	$w_i(t+1) = w_i(t) + x_i(t)$
if output 1, should be 0 (class B)	$w_i(t+1) = w_i(t) - x_i(t)$

Figure 2.4: Perceptron learning algorithm (Beale and Jackson, 1990).

The weights in the above algorithm are adjusted by a step size of 1. This can present problems: mainly that when it alters a weight it may also be affecting an already correct response to another type of input. Thus, there have been a number of modifications made to this basic learning rule. The simplest is to introduce a multiplicative factor of less than one into the weight adaption term. This reduces the effect of each weight change, forcing the network to make smaller steps towards the solution and is shown in Figure 2.5 (Beale and Jackson, 1990).

Adapt weights:

if correct	$w_i(t+1) = w_i(t)$
if output 0, should be 1 (class A)	$w_i(t+1) = w_i(t) + \eta x_i(t)$
if output 1, should be 0 (class B)	$w_i(t+1) = w_i(t) - \eta x_i(t)$

where $0 \leq \eta \leq 1$ represents a gain term that controls the adaption rate.

Figure 2.5: Modification to basic adaption rates (Beale and Jackson, 1990).

Another algorithm, suggested by Widrow and Hoff, used a similar idea. They realised that it would be better to change the weights more when the error was large and by smaller steps when the weighted-sum was close to what was required for a correct solution. They proposed a rule, called the *Widrow-Hoff delta rule* that calculates the difference between the weighted-sum and the required output, and calls this the *error*, denoted Δ. Therefore, the result is not passed through the step function while learning, only when actually being used for classification (Beale and Jackson, 1990). Figure 2.6 gives the new weight adaption rule, where $0 \leq \eta \leq 1$ is a gain function that controls the adaption rate.

Adapt weights using Widrow-Hoff delta rule:

$$\Delta = d(t) - y(t)$$

$$w_i(t+1) = w_i(t) + \eta \Delta x_i(t)$$

$$d(t) = \begin{cases} +1, & \text{if input from class A} \\ 0, & \text{if input from class B} \end{cases}$$

where $0 \leq \eta \leq 1$ represents a gain term that controls the adaption rate.

Figure 2.6: Widrow-Hoff delta rule (Beale and Jackson, 1990).

Another alternative is to use bipolar values, -1 and +1, instead of the binary values, 0 and +1 used so far. This has the advantage of allowing weights not used in the incorrect response, to be adapted as well, which can also speed up training (Beale and Jackson, 1990).

2.3.2 Limitation of the Perceptron

The problem with the above model and learning rule, as highlighted by Minsky and Papert, is that it is attempting to learn a linear separation between two classes of inputs and if the problem is not linearly separable by nature then it can not find a solution (Beale and Jackson, 1990). For example, in Figure 2.7 (a) there are two types of fish, the first, class A, are short but quite bulky the second, class B, is long and slender. If the network is given two inputs: weight and length of the fish, then it would learn, over many examples, the line separating the two classes as shown. Figure 2.7 (b), however, shows a graph of the simple XOR problem. Class A in this case occurs when both inputs are the same while class B should be identified when both inputs are different. However, looking at the graphical representation of them it is clear that you can not draw a *single* line that separates the two classes.

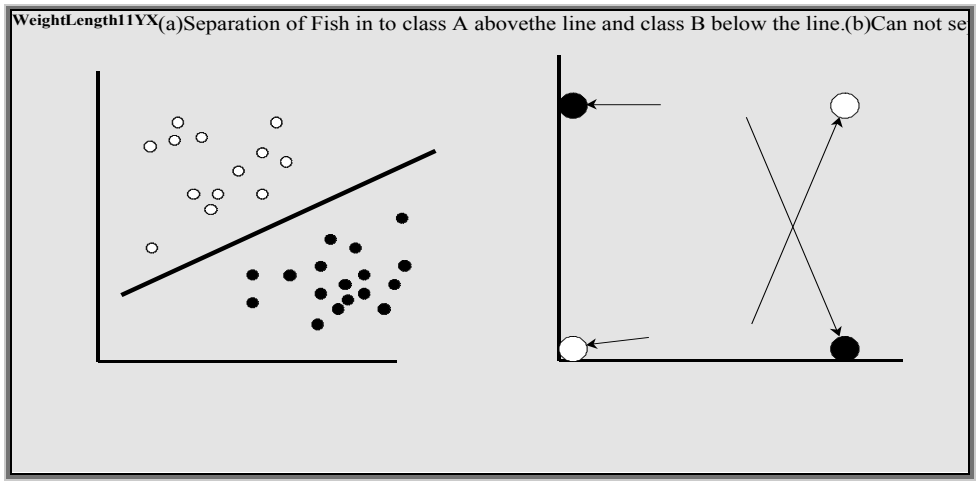


Figure 2.7: Examples of when the perceptron can and can not learn a solution.

2.4 The Multilayer Perceptron

One obvious solution to the XOR problem is to simply have two perceptrons each finding one linearly separable section then using a third to combine the results of the other two, as shown in Figure 2.8. However, while it is possible to set this by hand, the network cannot learn the weights by itself. This is because; when the output neuron receives an error it adjusts its inputs only and cannot change the inputs to the first two neurons, which also need updating. This occurs because there is no information available to the output neuron as to which inputs where ‘on’ or ‘off’, because the actual inputs are masked from the output units by the intermediate layers hard-limiting threshold function.

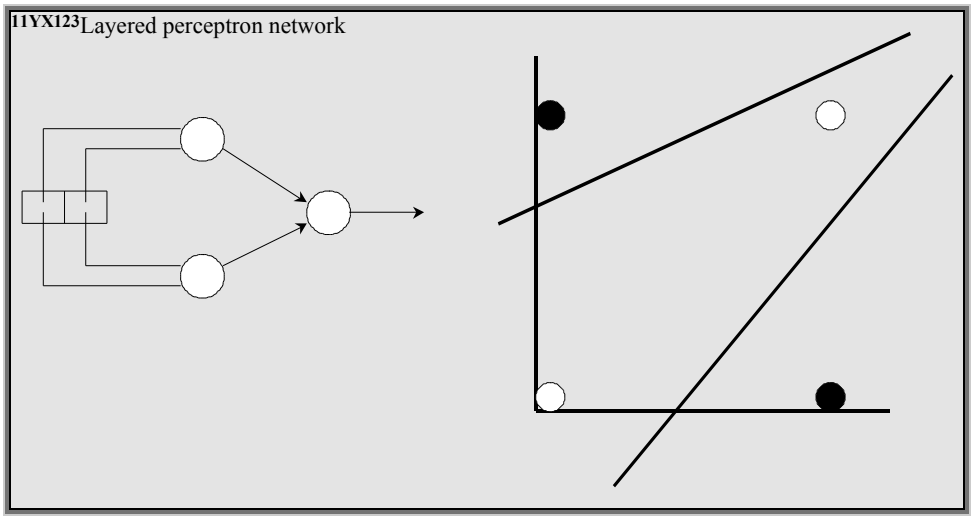


Figure 2.8: Multilayered perceptron network to solve XOR problem.

The solution then is to smooth the threshold function using a sigmoid thresholding function, or any other continuously differentiable monotonic function, as shown graphically in Figure 2.9, and defined mathematically in Equation 2.3, where $0 < f(net) < 1$ and k is a positive constant that controls the spread of the function: as $k \rightarrow \infty$ then $f(net) \rightarrow$ *the step function*. Therefore, generally it still turns on and off as before if there is a large difference between the threshold and the summation of inputs but if they are nearly identical then it will give a value in between the two extremes. This means we will have more information about when we need to strengthen or weaken the relevant weights (Beale and Jackson, 1990).

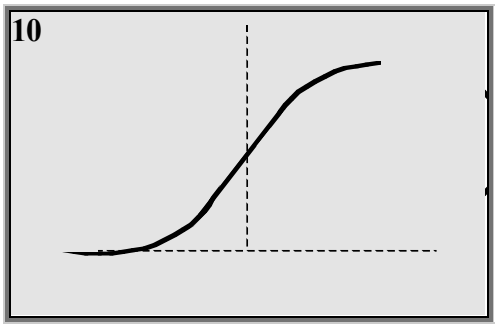


Figure 2.9: Sigmoid Threshold Function shown graphically.

$$f(net) = 1 \Big/ \left(1 + e^{-k\ net}\right)$$

Equation 2.3: The sigmoid function.

2.4.1 The New Model

A new model can now be created using this new type of artificial neuron called a *multilayer perceptron*. The new model has an input layer, an output layer and any number of hidden layers in between, as illustrated in the three-layer network shown in Figure 2.10. The neurons in the hidden layers and the output layer use the sigmoid threshold function instead of the step function used earlier. The input layer units serve only to distribute the inputs between each of the neurons in the first hidden layer and do not apply any thresholding function (Beale and Jackson, 1990).

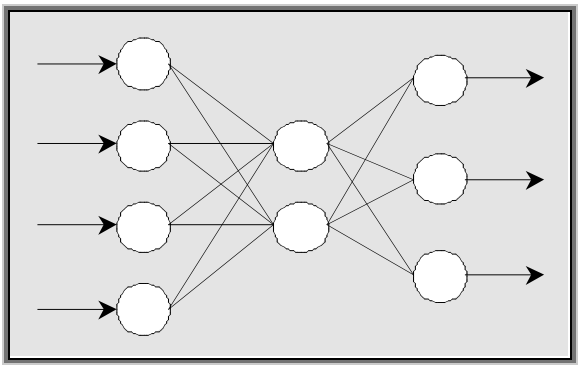


Figure 2.10: Three-layer example of the multilayer perceptron model.

2.4.2 The New Learning Rule

Due to the change in the thresholding function and the introduction of hidden layers the new model also requires a new learning rule. The basic rule suggested by Rumelhart, McClelland, and Williams in 1986 is called the *generalised delta rule*, or more commonly the *backpropagation rule*. The new rule operates in a similar way as the earlier ones did except in its method of updating weights (Beale and Jackson, 1990).

The backpropagation rule works by first presenting the network with input data representing a pattern. When the network has calculated its solution according to the random initial weights, a comparison is made between the network's answer at each output node and what the correct answer for that node should have been. An error value is calculated according to the difference using an error function. This error information is then used to adjust the weights of the connections leading to that node by a small amount in order to reduce the error. The error is also passed back to the previous hidden layer to allow that layer's nodes to adjust their weights as well (Beale and Jackson, 1990).

The error for a particular set of inputs, or pattern, denoted *p*, is calculated for some node *j* in the output layer using Equation 2.4, where t_{pj} represents the target output, o_{pj} is the actual output and δ_{pj} is the amount of error applied to that node. The function $f\_j(net_{pj})$ is the derivative of the thresholding function, applied to the activation at node *j*, which is simply the weighted sum of the inputs to that node, as used for the single perceptron.

$$\delta_{pj} = f'_j(net_{pj})(t_{pj} - o_{pj})$$

Equation 2.4: Error calculation for output node.

This function though does not work for hidden nodes because the correct target output is unknown. Therefore, the error at the output node is propagated back to the hidden nodes and used in Equation 2.5 instead (Beale and Jackson, 1990).

$$\delta_{pj} = f'_j(net_{pj})\sum_k \delta_{pk} w_{jk}$$

Equation 2.5: Error calculation for hidden nodes.

It was described earlier that any continuously differentiable monotonic function could be used as the thresholding function. However, the most commonly used is the sigmoid function mainly because it's derivative, as required in the above equations, is simple, making implementation easier (Beale and Jackson, 1990). Equation 2.6 shows the derivative of the earlier defined sigmoid function, while Equation 2.7 and Equation 2.8 give the above error calculations again with the threshold-function's derivative incorporated.

$$f'(net) = ko_{pj}(1 - o_{pj})$$

Equation 2.6: Derivative of the sigmoid function with respect to *f'(net)*.

$$\delta_{pj} = ko_{pj}(1 - o_{pj})(t_{pj} - o_{pj})$$

Equation 2.7: Error calculation for output node with *f'(net)* incorporated.

$$\delta_{pj} = ko_{pj}(1 - o_{pj})\sum_k \delta_{pk} w_{jk}$$

Equation 2.8: Error calculation for hidden node with *f'(net)* incorporated.

Both these functions provide the means to change the error function so as to be sure of reducing it, as it is passed back through the network. It may also be noticed that the error for the hidden nodes is proportional to the error in the nodes following it, such as the output nodes. Therefore, it is vital that the output nodes are calculated first, followed by the last hidden node and so on back to the first set of hidden nodes (Beale and Jackson, 1990). Hence, the learning rule being often referred to as *backpropagation*.

Finally, once we have the amount of error to be applied at each node the weights for each of the connections leading to the nodes can be adjusted using Equation 2.9, where η is a gain term, and w_{ij} is the weight from node i to node j at time t.

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_{pj} o_{pj}$$

Equation 2.9: Weight adaption formula using gain term for multilayer perceptrons.

2.4.3 Multilayer Perceptrons as Classifiers

It was mentioned earlier that the simple perceptron was limited to only being able to learn a single linear separation of data patterns. The purpose of the multilayer network then was to find a way of combining many of these in order to allow us to learn more complex shapes (Beale and Jackson, 1990). This can now be achieved with the use of hidden layers. For instance, a network with one hidden layer containing one node and an output layer with one node will still only give us the linear separation of a simple perceptron, however, if we have two hidden nodes the network would be able to find two linear separators. The output node would then act as a logical *AND*, combining the two and therefore, solve the XOR problem, as we wanted to achieve back in Figure 2.7. Of course we can add as many nodes into the first hidden layer as we wish, allowing even more complex shapes, see Figure 2.11.

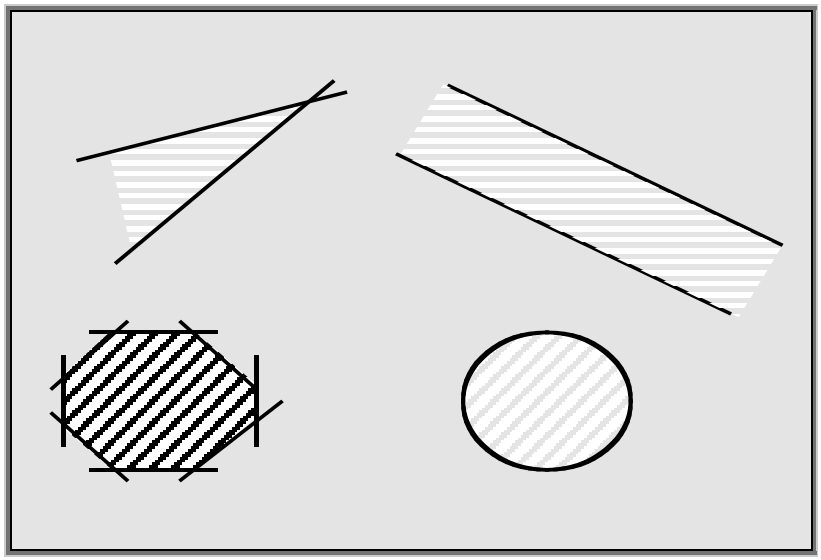


Figure 2.11: Examples of open (top two) and closed (bottom two) convex hulls.

The shapes that can be created by a single hidden layer are called *convex hulls*, and have the characteristic that any point within the convex hull can be connected to any other by a straight line that does not cross the boundary region. However, if we add another layer of hidden nodes then the units in this layer will be getting convex hulls from the first layer not single lines as the first layer did. Thus, the combination of many convex hulls can therefore, represent any arbitrarily complex shape we want and are capable of separating any classes, some examples are shown in Figure 2.12. Therefore, if four layers can represent any shape then there is no reason to ever have more layers (Beale and Jackson, 1990).

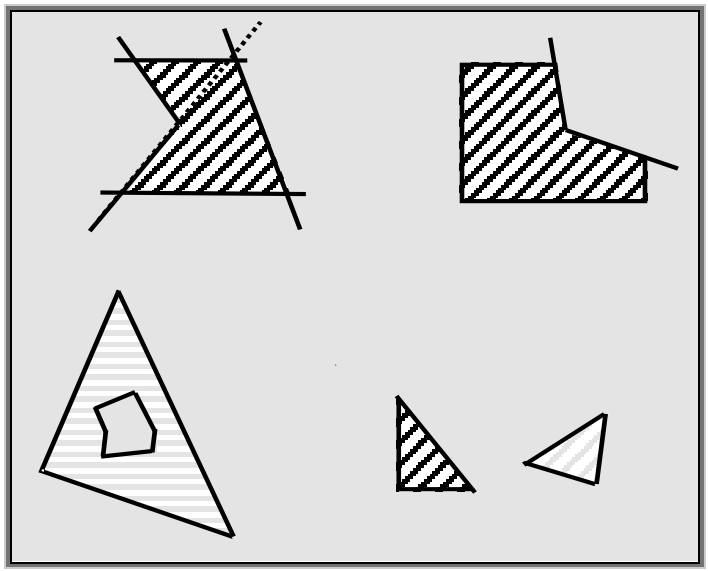


Figure 2.12: Examples of arbitrary regions formed by combining various convex hulls.

2.4.4 Problems with the Multilayer Perceptron

A major problem with the backpropagation algorithm is that it can sometimes find a stable solution that is incorrect. In such situations the network is said to have found a *local minimum*. Basically, the network learns by reducing the amount of error on each time step. However, it can sometimes reach a point where it cannot improve its solution any more without first making it worse (Beale and Jackson, 1990). Of course there is no way for the network to know this so it has to remain with the incorrect result already found. There have been a number of approaches to minimising these occurrences, which this section will briefly cover.

One common method used is to continually lower the gain term. By starting out with a large gain term η, large steps can be taken towards the correct solution avoiding many potential local minima. As the gain is decreased the network weights settle into a solution without overshooting the stable solution. This method can often bypass local minima at first and hopefully settle into a stable position in some deeper minima without oscillating wildly. However, the reduction in the gain term can slow learning (Beale and Jackson, 1990).

A second method sometimes used involves allowing the system to gather momentum that can then carry it over the lip of local-minima. This is done by adding an extra term into the weight adaption equation that produces a large change in the weight, if the changes are currently large and will decrease as the changes become less. This method can also speed up convergence along shallow gradients as the path downhill picks up speed (Beale and Jackson, 1990). The momentum term is shown in Equation 2.10, where α is the momentum factor.

$$\delta_p w_{ji}(t+1) = w_{ji} + \eta \delta_{pj} o_{pi} + \alpha\left(w_{ji}(t) - w_{ji}(t-1)\right)$$

Equation 2.10: Momentum term.

Another method is to increase the number of units within the hidden layer. This can help the internal representation between classes by providing a better recording of the inputs and lessen the occurrence of minima. Also, the addition of random noise can perturb the gradient descent algorithm from the line of steepest descent. This noise is often enough to knock the system out of local minima and has the advantage of not increasing computation, thus is not slower than the standard direct gradient descent algorithm (Beale and Jackson, 1990).

2.5 Conclusion

Artificial neural networks and neural computing is one of the most rapidly expanding areas of research in computing. It has also seen a number of important applications developed, especially since its resurgence during the mid 1980s. Essentially, an artificial neural network is an information processing system that has certain performance characteristics in common with biological neural systems (Fausett, 1994). One of the main features of neural networks, and the purpose behind their being used in conjunction with reinforcement learning, is their ability to generalize, that is, they can still classify sets of inputs even when those inputs have never previously been seen. They perform this generalization by identifying features of the input pattern that they have learnt to be significant. Thus a pattern is classified according to the similarity of its features to those of previously seen patterns (Beale and Jackson, 1990).

This chapter has provided a brief overview of the basic algorithms used in neural networks. It established a grounding in human neural biology, and covered early neural network models that where only capable of solving linearly separable problems. Finally, it discussed in some detail the multilayer perceptron model and basic backpropagation learning rule as these will be used during implementation. It has not covered many areas such as Kohonen self-organizing networks, Hopfield networks or Adaptive Resonance Theory, primarily because they are not being used in this thesis.

CHAPTER 3 Reinforcement Learning

Openings teach you openings. Endgames teach you chess!

Stephan Gerzadowicz

In the previous section a number of learning rules were studied, which all stemmed from the central concept of having a teacher or some set of known correct and incorrect results from which to learn. These *supervised learning* methods, however, are not always appropriate. An example of this would occur in areas where we do not already have perfect knowledge ourselves. Through observing young children it is clear that these *supervised learning* methods are not their primary form of learning. Instead, they utilize a direct sensori-motor connection to their environment, which produces a wealth of information about the consequences of actions, about cause and effect and, therefore, how to achieve particular goals or avoid dangers. Edward Thorndike describes this type of interaction with the environment, as learning through 'reinforcement' and that it forms the underlying foundation to learning and intelligence (Sutton and Barto, 1998).

Reinforcement Learning is a field of research that encompasses many computational techniques used to simulate Thorndike's theory to learning. However, these techniques themselves do not define the model but the class of problem the methods try to solve (Kaelbling et al, 1996). Therefore, any method that is well suited to solving such a problem is considered to be a reinforcement learning method (Sutton and Barto, 1998).

This Chapter, after providing a brief historical overview, will review the fundamentals of the reinforcement learning model and the components that it comprises. Secondly, it will look at some well-established reinforcement learning methods such as Dynamic Programming (DP), Monte Carlo (MC) and Temporal-Difference (TD) Learning. Finally, it will investigate some advanced aspects of reinforcement learning: namely, eligibility traces and integration of artificial neural networks for function generalization.

3.1 History of Reinforcement Learning

There are two primary threads of research that have led to what is now called reinforcement learning. The first thread comes from the problem of optimal control and its solution using value functions and dynamic programming. The second thread originated from research carried out within the field of the psychology of animal learning and concerns learning by trial-and-error. This was used in some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s (Sutton and Barto, 1998).

The term 'optimal control' first surfaced in the late 1950s and describes the problem of designing a controller to minimize a measure of a dynamic system's behaviour over time. Richard Bellman, through extending a nineteenth-century theory of Hamilton and Jacobi, developed the concept of a dynamic systems state and value function, This lead to his defining a functional equation often called the *Bellman equation*. The collection of methods that were developed for solving these types of optimal control problems is now known as dynamic programming (DP). Bellman also introduced another type of control problem known as Markovian decision processes (MDPs) that are a discrete stochastic version of DPs (Sutton and Barto, 1998).

The concept of trial-and-error learning began in psychology, where 'reinforcement' theories of learning are common. Edward Thorndike described trial-and-error learning, essentially to be the idea that actions followed by good or bad outcomes have their tendency to be reselected altered accordingly. Thorndike called this the 'law of effect', which has since been widely recognized as an obvious basic principle underlying much behaviour (Sutton and Barto, 1998).

Early work in artificial intelligence stemmed from this concept of trial-and-error learning. However, while neural network pioneers like Rosenblatt, Widrow and Hoff were clearly influenced by Thordike's theory through their use of terminology such as rewards and punishments but the systems they studied were actually supervised learning systems suitable for pattern recognition and perceptual learning. While these systems do use error information to adjust

weights and learn they do not exhibit selectional characteristics, which are an essential element to the concept of trial-and-error learning as described in the 'law of effect' (Sutton and Barto, 1998).

While there was some work done in 'true' trial-and-error learning it wasn't really revived until Harry Klopf realized that the essential aspects of adaptive behaviour were being lost by main stream supervised learning techniques. Klopf believed the missing element was the drive to achieve some result by controlling the environment toward some ends and away from undesired ends (Sutton and Barto, 1998).

These ideas by Klopf strongly influenced work by Sutton and Barto who developed them further into learning rules driven by changes in temporally successive predictions in the 1980s. Finally, Chris Watkins developed Q-learning, which fully integrated temporal difference learning and optimal control problems. It wasn't however, until Gerry Tesauro's backgammon playing program, TD-Gammon (4.3.3), in 1992, that reinforcement learning reached its peak in popularity (Sutton and Barto, 1998).

3.2 Reinforcement Learning Model

In its most fundamental state, the reinforcement-learning problem is one that involves an *agent* learning through interaction with its *environment* to achieve a goal or goals (Sutton and Barto, 1998). This section will first provide a brief overview of the major elements in reinforcement learning. Secondly, due to reinforcement learning using a system of return values over time to adapt to its environment, it is also necessary to discuss the different tasks used in the model. It will also address the issue of evaluative feedback and action selection. Finally, a brief overview of Markov Decision Processes (MDPs) is included due to their close correlation to the majority of reinforcement learning problems.

3.2.1 Reinforcement Learning Elements

Figure 3.1 shows the standard reinforcement-learning model, where the *agent* is connected to its environment via its perceptions and actions. The agent first receives the current state s_t of the environment and evaluates an appropriate action a_t by using its current *policy*. In the following time step the agent will then perceive the consequences of its action through a numerical reward r_{t+1} along with its next state s_{t+1}. The reward is determined through the application of both a *reward function* and a *value function*

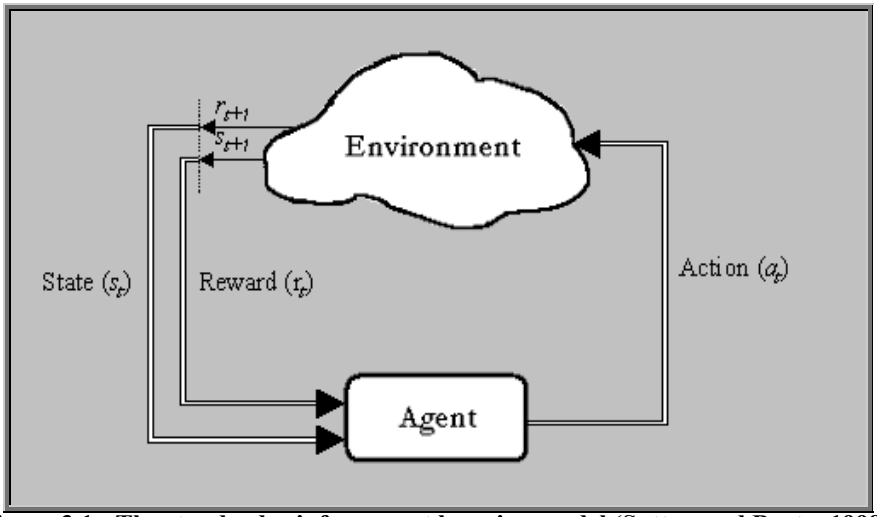


Figure 3.1: The standard reinforcement learning model (Sutton and Barto, 1998).

3.2.1.1 An Agents Policy

The *agent* determines the action to be taken at each time step by applying a mapping from perceived states to the probabilities of selecting each possible action when in those states. This mapping is called the agent's *policy*, denoted π_t, where $\pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$ (Sutton and Barto, 1998). Generally, an agent's policy can be thought of as its way of behaving to a particular situation and corresponds to what are called stimulus-response rules or associations in psychology (Op cit).

It is this mapping that reinforcement learning methods are attempting to optimize, by systematically altering the *policy* in such a way as to maximize the total amount of *reward* over the long run. This differs from the earlier mentioned *supervised learning* in that the agent is not told which action would have been in its best long-term interest (Kaelbling et al, 1996).

3.2.1.2 The Reward Function

A *reward function* defines the goal of the reinforcement-learning problem and determines the *immediate* reward response to an action. It can be thought of as a mapping of each state-action pair in the environment to a single number representing the desirability of the new state. Therefore, it defines what are good and bad events and is analogous to the experience of pleasure and pain in a biological system (Sutton and Barto, 1998).

Its is important to note, due to the reward function being the goal of the agent, that the agent itself should not be able to alter the mapping. Thus, the reward function is usually viewed as being outside the agent, and therefore, part of the environment. However, this does not prevent the agent from defining for itself a number of internal rewards.

It is also clear that if the agent learns to maximize its rewards then when it achieves this it must also reach its goal. Thus, the choice of rewards is very important. For instance, in chess we want the agent to learn how to win, which may or may not involve capturing pieces. However, if we were to allocate extra rewards for subgoals such as taking a piece then the agent may learn to take pieces at the cost of losing the game (Sutton and Barto, 1998).

3.2.1.3 The Value Function

The *value function* determines the total reward that a state can expect to accumulate over the future from the current state as opposed to the immediately expected reward that is given by the *reward function*. Therefore, a reward gives the immediate desirability of a state, while a value determines the long-term desirability of a state after considering the following states and their associated rewards (Sutton and Barto, 1998).

The use of value functions allows an agent a more in-depth analysis of its choices when determining its next action. For example, a state may have a low *reward* but it may lead to a more favorable outcome in the end, giving it a higher *value* function would lead to it being chosen above an action with a better reward but lower eventual value.

Values are in fact a prediction of likely rewards in the future and therefore, are secondary to rewards, as without rewards there would be no values. Nevertheless, values are of greatest concern when making and evaluating decisions as they maximize the total reward. However, they are also one of the most difficult aspects of reinforcement learning due to their need for being continually re-estimated from all observations made of the environment over the agents lifetime (Sutton and Barto, 1998).

3.2.1.4 The Model of the Environment

Recently, reinforcement-learning methods have also incorporated a model of the environment within the agent. These models attempt to mimic the behaviour of the environment for the purpose of *planning*. The introduction of these environmental models into the system make it possible to consider future situations before they occur and thereby, improve the agent's ability to decide on a course of action. For example, it may try to predict the resulting next state and reward (Sutton and Barto, 1998).

Early work in the field of reinforcement learning was purely trial-and-error based, however, it became clear that the methods used were closely related to *dynamic programming methods* and *state-based planning* methods, both of which use similar models (Sutton and Barto, 1998).

3.2.1.5 States

At each time step the agent, in a reinforcement learning model, receives some representation of the environment's state, where $s_t \in S$, where S is the set of all possible states. These states can take a wide variety of forms. For instance, they can be completely low-level sensation such as readings from sensors or they can be more high-level and abstract such as using symbolic representations of objects. A state can also be comprised of memory of past sensations and can even be entirely mental or subjective (Sutton and Barto, 1998). This means that systems can work on purely mental or computational problems by 'thinking' through a number stages or letting its 'mind wander' as its focus of attention changes.

3.2.2 Episodic and Continuing Tasks

A *task* in reinforcement learning is a complete specification of an environment and represents one instance of the reinforcement-learning problem. Sutton and Barto refer to two types of tasks: *episodic tasks* and *continuous tasks* in reinforcement learning problems. *Episodic tasks* have clearly definable subsequences, called *episodes*, within the problem. Episodes may consist of plays of a game, trips through a maze or any sort of repeated interactions. In such problems each episode ends with a special state called the *terminating state*. After each episode terminates, the system resets to a standard starting state or a sample from a standard distribution of starting states (Sutton and Barto, 1998).

However, not all problems have such naturally occurring and identifiable episodes but instead go on continually with no terminating state such as a continual processing-control task. These systems are called *continuous tasks* and can be more problematic in developing a return function as the final time step would be $T = \infty$ and the maximum total return could easily be infinite. Problems in this domain calculate the future reward from a particular state by defining a distance to look ahead and predicting the expected reward to that point. It does this by using a concept called *discounting*. In this approach a *discount rate* γ is selected, such that $0 \leq \gamma \leq 1$, and this is used to calculate the discount return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Equation 3.1: Calculation of a Discounted Return in Continuous Tasks.

An action a_t is then selected to maximize this *discount return* R_t rather than the expected reward as in episodic tasks. It can be seen in the above equation that if $\gamma = 0$ then $R_t = r_{t+1}$. Therefore, the agent is 'myopic' as it is only concerned with maximizing the immediate reward. If $0 < \gamma < 1$ and r_k is bounded, then the infinite sum will have a finite value. It is also clear that as $\gamma \rightarrow 1$ the agent becomes more farsighted, taking more future rewards into account.

It can be noted that although different names for the return value are used for each type of task, they amount to the same thing, a value judgement (prediction) for the proposed action.

3.2.3 Evaluative Feedback

One of the most important distinguishing features between reinforcement learning methods and other types of learning is that they *evaluate* the actions performed rather than being given the correct answer through *instruction*. It is this difference that allows for active *exploration* of the problem through *trial-and-error* search in order to discover good behaviour. This is because a purely evaluative feedback indicates how good the action taken is, but not whether it is the best or worst action possible. It is also the basis for methods of function optimization, such as Genetic Algorithms (GAs) (Sutton and Barto, 1998).

The notion of trial-and-error, though, indicates that the evaluative approach also introduces the issue of action selection, which has been a major area of research in reinforcement learning. The difficulty is that if we believe a particular action will generate a higher total reward than other, never tested actions, then our aim of maximizing reward would indicate that we should choose the higher valued action. However, some actions that may have yielded a much higher reward may remain untried. Also, situations can occur, where an action might normally be highly effective but the first time it is tried gives a small reward and so is given a low action-value and never tried again.

This problem highlights the need to find a balance between actions that *exploit* paths known to return high rewards, called *greedy* actions, and *exploratory* actions. The selection of the non-greedy action may cause a lower reward initially but improves the systems knowledge of the domain and will eventually result in higher returns when a more greedy approach is applied. This balancing act is often referred to as the ‘conflict’ between exploration and exploitation (Sutton and Barto, 1998).

Generally, whether it is better to explore or exploit depends on the precise values of the estimates, uncertainties and how many plays are remaining. There have been a number of methods formulated for performing this balancing act, some of which will be the focus of this section. However, many tend to make assumptions about prior knowledge that are either violated or impossible to verify in the full reinforcement learning problem.

3.2.3.1 Estimating Action Values

Before an action can be selected we first need some notion of each action's value. One of the simplest and natural methods of estimating an action's 'true' value is the *sample-average* method. It works by averaging the reward received when that action has been selected. For instance, if at the t^{th} play, action a has been selected k_a times prior to t and yielded rewards $r_1, r_2,\ldots, r_{ka}$ then its value is estimated to be $Q_t(a)$ using Equation 3.2, where $Q_t(a)$ represents the estimated value of action a at time t.

$$Q_t(a) = \frac{r_1 + r_2 + \mathrm{K} + r_{k_a}}{k_a}$$

Equation 3.2: Averaging rewards received for a particular action.

However, prior to the action being selected for the first time, namely $k_a = 0$, then $Q_t(a)$ is given a default value. It can also be observed that as $k_a \rightarrow \infty$, then by the 'law of large numbers' $Q_t(a)$ approaches the 'true' value of the action, denoted $Q^*(a)$ (Sutton and Barto, 1998).

This equation, however, indicates that the return values have been stored after each action selection, which is clearly an unnecessary waste of memory. Equation 3.3 gives a similar equation that offers a progressive method of calculating the average after each action selection. Basically, it takes the average estimate prior to the action being selected; denoted Q_k, where k is the number of rewards received, and adds a correction value for the new average. The correction is the difference between the new (r_{k+1}) and the old (Q_k) estimates, called the *error*, multiplied by the inverse of the number of values being averaged ($k+1$), referred to as the step-size.

$$Q_{k+1} = Q_k + \frac{1}{k+1}\left[r_{k+1} - Q_k\right]$$

Equation 3.3: Progressive update formula for calculating averages.

It can be noted that the step-size parameter is clearly going to decrease with each new estimate added to the average and will in fact approach 0 as $Q_k \rightarrow Q^*(a)$. The step-size parameter is often denoted by α, or more generally $\alpha_k(a)$. For example, the above incremental implementation of the sample-average method can also be described by Equation 3.4.

$$Q_{k+1} = Q_k + \alpha\left[r_{k+1} - Q_k\right]$$

$$\text{where } \alpha_k(a) = \frac{1}{k_a}$$

Equation 3.4: Progressive update formula with step-wise parameter.

The above sample-average method of estimating rewards is appropriate for stationary environments. However, when a non-stationary problem, where the true value we are trying to estimate is changing over time, is encountered a different approach is often required. In such a situation it makes sense to weight recent rewards more heavily than earlier ones. This can be implemented in a number of ways but the simplest and most popular is the *recency-weighted average* method. This method uses Equation 3.4 but applies a constant step-wise parameter where $0 < \alpha \leq 1$.

3.2.3.2 Simple Action Selection

Now, by using one of the above methods, we have our estimates for each of the possible actions, but given this information, which action do we select? The simplest action selection rule is obviously to select the action with the highest expected reward. This method will always exploit current knowledge to maximize immediate rewards and never considers any apparently inferior actions that may be better. This is clearly flawed. A simple alternative is to make this greedy selection most of the time but occasionally, say with probability ε, select an action at random, ignoring action values. Methods using this near-greedy action selection are called *ε-greedy* methods (Sutton and Barto, 1998).

An advantage with these methods is that as the number of plays increases every action will be sampled repeatedly even if initially they offer very low rewards. This allows a much broader analysis in a noisy system. However, if there is little or no variance in the returned values it may be better to initially select every action once, giving them their ‘true’ value, and then simply applying the purely greedy method as it would already know which action is the best removing the need for further search (Sutton and Barto, 1998).

3.2.3.3 Softmax Action Selection

While the ε-greedy method is an effective means of balancing exploration and exploitation, it does have one drawback, in that there is an equal chance that it will select the worst action as the next best. The obvious solution to this is to grade the probabilities so that the second best has a better chance of being selected than the worst. Methods using this approach are referred to as softmax action selection rules. The most common softmax method uses a Gibbs, or Boltzmann, distribution, given in Equation 3.5. This method gives the probability of selecting action a on the t^{th} play (Sutton and Barto, 1998).

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

Equation 3.5: Gibbs Distribution.

The parameter τ is a positive number called the *temperature*. High temperatures cause the set of probabilities to be nearly equal, therefore, strongly favoring exploration over exploitation. Low temperatures create greater difference in the probabilities and as $\tau \rightarrow 0$ the softmax method becomes the same as the greedy action selection (Sutton and Barto, 1998).

3.2.3.4 Initial Values

All the methods discussed so far are dependant on some initial action-value estimates, $Q_0(a)$.Therefore, these methods can be thought of as being biased by their initial estimates. In the sample-average method described first this bias disappears after the action has been selected once. However, for the methods using a constant α parameter this bias is permanent, though it does decrease its influence over time (Sutton and Barto, 1998).

This bias does not generally cause any harmful effects but it can offer a useful and simple means of encouraging exploration during the earlier stages of learning by giving optimistic values. For instance, by giving an initial value above the amount of reward that would be received by any of the actions then the agent will be 'disappointed' regardless of its choice of action and will explore other actions before returning to try it again (Sutton and Barto, 1998).

3.2.3.5 Reinforcement Comparison

Reinforcement Comparison methods take a different approach to the above action-value techniques. Primarily they look more directly at the fundamental issue underlying reinforcement learning, which is that actions followed by large rewards should be more likely to occur than those with small ones (Sutton and Barto, 1998). The question here is, how does the agent know what is a high reward?

In order to make such a judgement each reward must be compared to some standard or reference level, called the *reference reward*, denoted $\bar{r}$. A common choice for this reference reward is an average of all previously received rewards using a similar process as used in Equation 3.4. The step-wise parameter _ in this situation should be constant in the range $0 < \_ \leq 1$ as in the *recency-weighted average* method because distribution of rewards is changing over time as the policy changes (Sutton and Barto, 1998).

Given a reference to compare our rewards we must now select an appropriate action. Reinforcement comparison methods maintain a separate measure of their *preference* for each action, denoted $p_t(a)$. These preferences can be used in action selection in the same way as action-values were in the above methods, such as using the softmax method. After an action selection is made the preference is updated, by the difference between the reward, r_t, and the reference reward, using Equation 3.6, where _ is the step-wise parameter (Sutton and Barto, 1998).

$$p_{t+1}(a_t) = p_t(a_t) + \beta\left[r_r - \bar{r}_t\right]$$

Equation 3.6: Action preference update formula.

Reinforcement comparison methods can be very effective and sometimes outperform action value methods (Sutton and Barto, 1998). They are also the precursor to actor-critic methods detailed later in section 3.3.3.4.

3.2.3.6 Pursuit Methods

The final class of learning methods, called *pursuit methods*, being considered in this section is really an amalgamation of reinforcement comparison and action-value techniques. Basically, they maintain both action-value estimates and action-preferences, with the preferences continually being updated to pursue the greedy action as stipulated by the action value estimates. In the simplest of pursuit methods the action preferences are the probability, denoted $\pi_t(a)$, that action a is selected on play t (Sutton and Barto, 1998).

Thus, after each play all of the probabilities are updated to make the greedy action more likely to be selected. Firstly, the greedy action is incremented a fraction towards 1 using Equation 3.7, where $a^*_{t+1}=arg\ max_a\ Q_{t+1}(a)$ is the greedy action (or a randomly selected action from the set of greedy actions if there is more than one) for the $(t+1)^{st}$ play and _ is the step-wise parameter.

$$\pi_{t+1}(a^*_{t+1}) = \pi_t(a^*_{t+1}) + \beta[1 - \pi_t(a^*_{t+1})]$$

Equation 3.7: Greedy action update formula in simple pursuit method.

Secondly, all the other probabilities are decremented towards zero using Equation 3.8. Finally, the action values, $Q_{t+1}(a)$, are updated using one of the methods mentioned earlier.

$$\pi_{t+1}(a) = \pi_t(a) + \beta[0 - \pi_t(a)] \qquad \forall\, a \neq a^*_{t+1}$$

Equation 3.8: Non-greedy action update formula in simple pursuit method.

3.2.4 Markov Decision Processes

The field of reinforcement learning is strongly built around Markov Decision Processes (MDPs) in optimal control. While all MDP problems can also be thought of as reinforcement learning problems the reverse is only true if the *Markov property* holds. For a problem to be defined as having the Markov Property its *state* information must contain all relevant information including summaries of past sensations. Also if the environment has the Markov property then we can predict accurately the next state and expected return from the current state. Therefore, we could predict all the future states and rewards from the current position (Sutton and Barto, 1998).

For instance, the chess environment fits easily into the Markov property because the current state (position of the pieces) summarizes everything important about the complete sequence of positions that led to it. Of course, the precise moves that were taken have been lost, but all that really matters for the future of the game is still present. Also, it is theoretically possible, although computationally intractable, to predict all future states and associated rewards (Sutton and Barto, 1998).

The Markov property is important to reinforcement learning because, like MDPs, the reinforcement-learning problem assumes its decisions and values to be functions only of the current state. Therefore, the Markov case helps us to understand the behaviour of the algorithms and the algorithms can be successfully applied to many tasks with states that are not strictly Markov.

Earlier, it was mentioned that solving a reinforcement-learning task involves finding an optimal policy that achieves a lot of reward over the long run. For *finite MDP*, an MDP where the state and action spaces are finite, it is possible to precisely define an optimal policy. This is achieved by value functions defining a partial ordering over the set of policies by solving the Bellman optimality equation (3.3.1). Therefore, a policy π is defined to be better than or equal to a policy π' for all states if its expected reward is greater than or equal to that of π'. Such a policy is called an *optimal policy*. There can be one or many of these optimal policies and are denoted by π^* (Sutton and Barto, 1998).

If we could generate this partial ordering of policies for a given task then the agent would exhibit excellent performance. However, even if we have a complete and accurate model of the environment's dynamics it is rarely possible to achieve this level of optimality because most of the tasks of interest to us would involve extreme computational cost, both in processing power and memory. Therefore, for most reinforcement learning problems we are interested in finding approaches to approximately solving MDPs (Sutton and Barto, 1998).

3.3 Reinforcement Learning Methods

There have been a number of reinforcement learning methods created and they cannot all be covered here. However, many can fit into one of three broad fundamental classes of methods, which will be the focus of this section. The first, *Dynamic Programming* covers problems with a complete and accurate, Markovian model of the environment and are very well developed mathematically. The second called *Monte Carlo*, do not require a model and are conceptually simple but are not well suited to step-by-step incremental computation. The final class of methods, called *Temporal Difference*, could be seen as an amalgamation of the first two. It also does not require a model and is fully incremental but is far more complex (Sutton and Barto, 1998).

3.3.1 Dynamic Programming

Dynamic Programming (DP) refers to a group of algorithms that can compute optimal policies given a perfect model of the environment as in an MDP. Generally, DP algorithms have limited application in reinforcement learning, but they do represent what we are trying to approximate, just with less computation and without assuming a perfect model. The key concept in DPs is the use of value functions to organize and structure a search for good policies, as it generally is in reinforcement learning (Sutton and Barto, 1998).

In a finite MDP we assume that the set of states, represented by S, and actions, denoted $A(s)$, where $s \in S$, are finite. Also it is assumed that its dynamics are given by the transition probabilities $P^a_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$, and the expected immediate rewards $R^a_{ss'} = E\{r_{t+1} \mid a_t = a, s_t = s, s_{t+1} = s'\}$, $\forall\ s \in S, a \in A(s),$ and $s \in S^+$, where S^+ is S plus a terminal state if the problem is episodic (Sutton and Barto, 1998).

An optimal policy is a simple ordering of policies based on their optimal value functions, denoted by V^* or Q^*, that satisfy the Bellman optimality equations (Equation 3.9). In DP algorithms this is achieved by converting these general equations into update rules for improving approximations of the desired value functions (Sutton and Barto, 1998).

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] \quad or$$

$$Q^*(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s',a') \right]$$

$$\forall \; s \in S, a \in A(s), \text{and } s' \in S^+.$$

Equation 3.9: Bellman optimality equations.

This paper will look at the three primary algorithms for finding an optimal policy in Dynamic Programming. The first, called *policy iteration*, converges in only a few iterations. However, this algorithm is exponential in relation to the number of policies (Kaelbling et al, 1996). The second algorithm, called *value iteration*, reduces some of the time complexity of the first algorithm and the final method, called *asynchronous dynamic programming* improves flexibility (Sutton and Barto, 1998).

3.3.1.1 Policy Iteration

The Policy Iteration algorithm can be broken into three stages as illustrated in Figure 3.2.

1. Initialization
$V(s) \in R$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$

2. Policy Evaluation
Loop
$\Delta \leftarrow 0$
for each $s \in S$ do
$v \leftarrow V(s)$
$V(s) \leftarrow \sum_{s'} P^{\pi(s)}_{ss'} \left[R^{\pi(s)}_{ss'} + \gamma V(s') \right]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
policy_stable $\leftarrow$ true
for each $s \in S$ do
$b \leftarrow \pi(s)$
$\pi(s) \leftarrow \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$
If $b \neq \pi(s)$, then *policy_stable* $\leftarrow$ false
If *policy_stable*, then stop else go to 2

Figure 3.2: Policy iteration using iterative policy evaluation (Sutton and Barto, 1998)

Firstly, starting values are given for each reward-value and policy. Secondly, the state-value function V^π for all π, called the *policy evaluation*, is calculated. To produce each successive approximation, V_{k+1} from V_k, policy evaluation performs the same operation to every state s. This is achieved by replacing the old value for s with a new value calculated from the old values of all successor states of s and the expected immediate reward along all the

one-step transitions possible under the policy being evaluated (Sutton and Barto, 1998). This operation is called a *full backup* and can be expressed diagrammatically as in Figure 3.3.

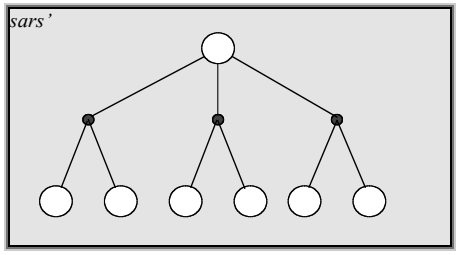


Figure 3.3: Example of a backup diagram for V^{π}

Therefore, informally the *value* of state *s* under policy π, denoted $V^{\pi}(s)$, is the expected return when starting in *s* and following π thereafter. The evaluation loop potentially can continue infinitely, therefore, it is halted when the effect of each sweep is small. The main reason for computing the value function for each policy is to help find better policies.

The third stage, steps through each state determining whether the policy π at state *s* should be changed. A change would be made if it were determined that selecting a different action improves the state-value function for the policy in a particular state (Sutton and Barto, 1998). Basically, after a policy, π, has been improved using V^{π} to yield a better policy π', we can then calculate V^{π} again yielding an even more improved policy π''. Therefore, a sequence of policy and value function improvements can be constructed (Figure 3.4), where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement (Sutton and Barto, 1998).

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \text{L} \xrightarrow{I} \pi^* \xrightarrow{E} V^*.$$

Figure 3.4: Sequence for improving policies and value functions in policy iteration algorithm.

The key here is that each iteration is guaranteed to improve a policy over the previous one (unless already optimal), see Sutton and Barto (page 95) for the proof of the policy improvement theorem. Therefore, due to the finite MDP having a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite amount of iterations (Sutton and Barto, 1998).

3.3.1.2 Value Iteration

One major drawback with the policy iteration algorithm is that each sweep through involves a protracted iterative policy evaluation stage requiring multiple sweeps through the state set. Also, policy convergence only occurs when V^{π} converges exactly to the limit, which may be well after an optimal solution is already being exhibited (Sutton and Barto, 1998).

Value iteration is a method that truncates policy evaluation after just one sweep (one backup of each state) and therefore, reduces the time complexity. Interestingly, this does not lose the convergence guarantee of policy iteration. This is achieved by using a simple backup operation, using Equation 3.10, which combines both policy improvement and truncated policy evaluation steps (Sutton and Barto, 1998).

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]$$

Equation 3.10: Value iteration formula, combining policy improvement and truncated policy evaluation steps

Value iteration, as in policy iteration, converges to the limit making it require an infinite number of iterations. Therefore, termination is achieved when the change in the value function in the last sweep is small. This can be seen in Figure 3.5, which gives the full value iteration algorithm. The algorithm effectively combines one sweep of policy evaluation and one sweep of policy improvement into each of its sweeps. Convergence, while still guaranteed to occur, is slower than policy iteration but can be improved by combining the two approaches (Sutton and Barto, 1998).

Initialise $V(s)$ arbitrarily for all $s \in S^+$

Loop
  $\Delta \leftarrow 0$
  for each $s \in S$ do
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V(s') \right]$

Figure 3.5: Value iteration algorithm (Sutton and Barto, 1998).

3.3.1.3 Asynchronous Dynamic Programming

The primary drawback with the two above methods is that they involve iterative operations over the entire state set of the MDP. However, in many problems the state set can be huge and a single sweep can be prohibitively expensive. Asynchronous dynamic programming methods provide a means of avoiding long sweeps before making policy improvements. This is achieved by backing up the values of states in any order whatsoever, using whatever values are available for other states. Therefore, it is possible for some states to be backed up several times before another is backed up the first time (Sutton and Barto, 1998).

However, in order to converge to an optimal solution all states must still be backed up but this method does allow more flexibility in selecting states. This flexibility makes it easier to intermix computation with real-time interaction by focusing the DP algorithm's backups onto the elements of the state set that are most relevant to the agent at that time (Sutton and Barto, 1998).

3.3.2 Monte Carlo Methods

Generally, the complexity of many problems in reinforcement learning or the lack of a perfect model of the environment rules out dynamic programming methods. The aim of reinforcement learning is to try to simulate the DP algorithm without incurring its computation costs and without being limited to a Markovian model (Sutton and Barto, 1998).

This section investigates a collection of methods, referred to as *Monte Carlo*, which only require past experience, such as sample sequences of states, actions, and rewards generated from *on-line* or *simulated* interaction with an environment, to improve behaviour. Learning from on-line experience requires no prior knowledge of the dynamics of the environment and can still potentially achieve optimal behaviour. Learning from simulation experience can also be very powerful but does require a model capable of generating sample transitions, but not the complete probability distribution of a Markovian model (Sutton and Barto, 1998).

Monte Carlo methods are a means of solving reinforcement learning problems by averaging sample returns and are generally only used on episodic tasks as these provide a well-defined return. It is only on the completion of an episode that value estimates and policies are changed, making this is an incremental approach in an episode-by-episode sense (Sutton and Barto, 1998). This section will briefly consider a selection of methods within this class.

3.3.2.1 Estimating State Values

Recall that policy evaluation for DPs first calculated a value for each state and then used these to select an optimal greedy policy. The value of a state is the expected return when starting at that state. In Monte Carlo methods we can obviously estimate this state value from experience by simply averaging the returns observed when that state has been visited (Sutton and Barto, 1998).

The two primary algorithms used are the *first-visit MC method* and the *every-visit MC method*. The first method averages the returns achieved after that state was reached the first time during a particular episode. This is the most common and the algorithm is given in Figure 3.6. The second averages the returns achieved after every visit to that state. While neither algorithm ever terminates, due to the law of large numbers the average of these estimates converges to their expected value (Sutton and Barto, 1998).

Initialise:
$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state- value function
Returns(s) $\leftarrow$ an empty list for all $s \in S$

Loop forever:
(a) Generate an episode using π
(b) For each state *s* appearing in the episode
$R \leftarrow$ return following the first occurrence of *s*
Append *R* to *Returns(s)*
V(s) $\leftarrow$ average (*Returns(s)*)

Figure 3.6: First-visit MC method for estimating V^{π} (Sutton and Barto, p113, 1998)

3.3.2.2 Estimating Action Values

When a model is available calculating state values is sufficient in determining a policy. For instance, you simply select the action that leads to the best reward and next state. However, if we do not have a model then state values are not enough. In addition a method for determining which is the best action is also required. This involves the estimation, $Q^{\pi}(s, a)$, of expected returns when selecting action a, from state s and following policy π thereafter. This is essentially the same process as the methods used in estimating state values, except we are estimating state-action pairs, and the same methods can be applied (Sutton and Barto, 1998).

There is, however, one complication in that many relevant state-action pairs may never be visited and, therefore, will never improve with experience. To correctly compare alternatives we must have estimates of all the actions from each state not just the favored ones. This is the problem of *maintaining exploration*, as discussed earlier. For policy evaluation to work correctly for action values then this process of exploration must be continual. One way of doing this is by specifying that the first step of each episode starts at a state-action pair, and that all of these pairs has a nonzero probability of being selected from the start. This ensures that all pairs are visited an infinite number of times in the limit of an infinite number of episodes. This is called the assumption of *exploring starts* (Sutton and Barto, 1998).

3.3.2.3 Monte Carlo Control

Now that we have estimates of both state values and state-action pairs we can use them to approximate optimal policies. Essentially, this can be done following the same pattern as with dynamic programming. Therefore, policy improvement is achieved by making the policy greedy with respect to the current *value function* or *action-value* function if there is no model. However, we must make two assumptions in order to obtain a guarantee of convergence to an optimal policy. The first is that every episode uses exploring starts and the second is that policy evaluation operates over an

infinite number of episodes. However, neither allows for a practical algorithm and will therefore, need to be removed (Sutton and Barto, 1998).

First, let us consider the assumption of an infinite number of episodes. This is not a new problem and was handled earlier in the DP algorithms simply by taking a measure of the error value and terminating when that value was sufficiently small. However, in Monte Carlo methods, with the exception of trivial problems, this will require far too many episodes. Another method is to forgo completing policy evaluation before returning to policy improvement. Taking this idea to the extreme results in the value iteration method, discussed earlier, in which only one iteration of policy evaluation is performed between each step of policy improvement (Sutton and Barto, 1998).

This method of alternating between evaluation and improvement is also intuitively natural when applied on an episode-by-episode basis. For instance, when an episode is complete, it makes sense to use the returns and perform policy improvement and then immediately perform policy improvement on all the states visited in the episode. Figure 3.7 shows this algorithm, called *Monte Carlo ES*, where *ES* stands for *Exploring Starts* (Sutton and Barto, 1998).

Initialise, for all $s \in S$, $a \in A(s)$**:**
$Q(s, a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s, a) \leftarrow$ empty list

Loop forever:
(a) Generate an episode using exploring starts and π
(b) For each pair *s, a* appearing in the episode
$R \leftarrow$ return following the first occurrence of *s, a*
Append *R* to *Returns(s, a)*
$Q(s, a) \leftarrow$ average (*Returns(s, a)*)
(c) For each s in the episode:
$\pi(s) \leftarrow \arg\max_a Q(s, a)$

Figure 3.7: A Monte Carlo algorithm assuming exploring starts (Sutton and Barto, p120, 1998).

3.3.2.4 On-policy Control

The assumption of exploring starts cannot be relied on in the general case particularly when learning from real interactions with the environment. Therefore, a method is needed to ensure that actions are selected infinitely often other than exploring starts. There are two approaches to do this, resulting in the methods *on-policy* and *off-policy*. This section will concentrate on on-policy control methods (Sutton and Barto, 1998).

On-policy methods attempt to evaluate or improve the policy that is used to make decisions. The control policy is not necessary the greedy policy but instead is a policy that is gradually shifted toward the deterministic optimal policy, and is often called a soft policy. This is generally achieved through the application of one of the methods similar to those discussed earlier in section 3.2.3, such as *ε-greedy* policy selection, referred to in this sense as *ε-soft* (Sutton and Barto, 1998). Figure 3.8 illustrates an *ε-soft* on-policy algorithm.

Initialise, for all $s \in S, a \in A(s)$**:**
$Q(s, a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary ε*-soft* policy
Returns(s, a) $\leftarrow$ empty list

Loop forever:
(a) Generate an episode using π
(b) For each pair *s, a* appearing in the episode
$R \leftarrow$ return following the first occurrence of *s, a*
Append *R* to *Returns(s, a)*
$Q(s, a) \leftarrow$ average (*Returns(s, a)*)
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s, a)$
For all $a \in A(s)$:

$$\pi(s, a) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon / |A(s)| & \text{if } a = a^* \\ \varepsilon / |A(s)| & \text{if } a \neq a^* \end{cases}$$

Figure 3.8: An ε-soft on-policy Monte Carlo control algorithm (Sutton and Barto, 1998).

3.3.2.5 Off-Policy Control

On-policy methods estimate the value of a policy while using it for control. This policy can also be called the *behaviour* policy. In Off-policy, however, the two operations of estimation and behaviour are separated. Therefore, the policy used to generate the behaviour may in fact be different from the policy being evaluated and improved, called the *estimation* policy. The advantage of this approach is that the estimation policy could be greedy while the behaviour policy can continue to sample all possible actions (Sutton and Barto, 1998).

In order for this to be achieved we require that every action taken under our estimation policy, π, has at least some possibility of being taken under our behaviour policy π'. That is, we require that if $\pi(s, a)>0$ then $\pi'(s, a)>0$ as well. Therefore, we require that the behaviour policy to be soft (Sutton and Barto, 1998).

This method works by calculating the probabilities, $p_i(s)$ and $p_i'(s)$, of the complete sequence of states and actions occurring after the i^{th} first visit to state s for both π and π' respectively. The observed return from this sequence is denoted $R_i(s)$. Now in order to calculate an estimate $V^{\pi}(s)$, for the value of state s, we must average the weight of each return by the relative probability of the sequence occurring under π and π', that is, $p_i(s)/p_i'(s)$ (Sutton and Barto, 1998). Therefore, over $n_{s,}$ observed sequences the estimated value of V(s) is given by Equation 3.11.

$$V(s) = \frac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p_i'(s)} R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p_i'(s)}}$$

Equation 3.11: Estimating the value of state s in off-policy control.

However, the probabilities $p_i(s)$ and $p_I'(s)$are usually unknown in Monte Carlo methods. Thus, normally just their ratio is used, which requires no knowledge of the environment's dynamics (Sutton and Barto, 1998). Equation 3.12 illustrates how this ratio can be determined entirely from the

two policies π and π', where $T_i(s)$ is the time of termination of the i^{th} episode involving state s.

$$\frac{p_i(s_t)}{p_i'(s_t)} = \frac{\prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k) P^{a_k}_{s_k s_{k+1}}}{\prod_{k=t}^{T_i(s)-1} \pi'(s_k, a_k) P^{a_k}_{s_k s_{k+1}}} = \prod_{k=t}^{T_i(s)-1} \frac{\pi(s_k, a_k)}{\pi'(s_k, a_k)}.$$

Equation 3.12: Calculation of probability ratio for policies π and π'.

An algorithm using this technique is illustrated in Figure 3.9. The primary problem with the algorithm is its tendency to learn *tails* of episodes. Where the *tail* is the sequence that occurred after the last non-greedy action selected. Less frequent selection of non-greedy actions increases the length of the tail, thereby improving learning. However, early portions of long episodes can still suffer from very slow learning (Sutton and Barto, 1998).

Initialise, for all $s \in S$, $a \in A(s)$:

$Q(s, a) \leftarrow$ arbitrary
$N(s, a) \leftarrow 0$; Numerator and
$D(s, a) \leftarrow 0$; Denominator of $Q(s, a)$
$\pi \leftarrow$ an arbitrary deterministic policy

Loop forever:

(a) Select a policy π' and use it to generate an episode:
$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
(b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
(c) For each pair s, a appearing in the episode at time τ or later:
$t \leftarrow$ the time of first occurrence of s, a such that $t \geq \tau$
$w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$
$N(s, a) \leftarrow N(s, a) + wR_t$
$D(s, a) \leftarrow D(s, a) + w$
$Q(s, a) \leftarrow \frac{N(s,a)}{D(s,a)}$
(d) For each $s \in S$:
$\pi(s) \leftarrow \arg\max_a Q(s, a)$

Figure 3.9: An off-policy Monte Carlo control algorithm.

3.3.3 Temporal-Difference Learning

Temporal-difference learning (TD) can be seen as a combination of dynamic programming and Monte Carlo ideas and is central to reinforcement learning. It is capable of learning from experience, as in Monte Carlo methods, without the need of a model of the environment's dynamics. Like dynamic programming, temporal-difference learning updates estimates based in part on other learned estimates, without the need for waiting for a final outcome (Sutton and Barto, 1998).

This process, called *bootstrapping* as in DPs, also means that temporal difference can be viewed as an incremental approach in a step-by-step sense, as opposed to the Monte Carlo episode-by-episode. This section will first look at methods of policy evaluation or the prediction problem of estimating the value function for a given policy. It will then investigate the problem of finding an optimal policy or control in TD (Sutton and Barto, 1998).

3.3.3.1 TD Prediction

TD is similar to Monte Carlo in that it also uses experience to solve the prediction problem. The essential difference is that Monte Carlo must wait until the end of the episode before it updates its estimates $V(s_t)$, whereas, TD methods need only wait until the next time step. At time t+1 they immediately form a target and make a useful update using the observed reward r_{t+1} and the estimate $V(s_{t+1})$. The simplest TD method, known as TD(0) is shown in Equation 3.13 and the algorithm is illustrated in Figure 3.10.

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

Equation 3.13: Estimation of state value for TD(0).

Monte Carlo methods calculate the target as an estimate of the expected value by using sample returns because the real value is not known. DP methods also use an estimate but not because the expected value is unknown (it is provided by the model). Instead, it is an estimate because $V^{\pi}(s_{t+1})$ is not known and $V_t(s_{t+1})$ is used as the current estimate. TD is seen as a combination of these two methods because it does both; it samples the expected value and it uses the current estimate V_t instead of the true value

V^{π}. Therefore, TD combines the sampling of Monte Carlo and the bootstrapping of DP (Sutton and Barto, 1998).

Initialise $V(s)$ arbitrarily, π to the policy to be evaluatedRepeat for each episode:Initialise sR

$\leftarrow$ π

$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$

$\leftarrow$

Figure 3.10: Algorithm for estimating V^{π} in TD(0).

3.3.3.2 Sarsa: On-Policy Control

As with the Monte Carlo methods there must be a balance between exploration and exploitation. When using TD prediction for the control problem, we find that methods can be placed in one of two classes: on-policy and off-policy (Sutton and Barto, 1998).

Similar to MC, we are interested in learning state-action pairs along with the state-value function. Therefore, we must estimate $Q^{\pi}(s, a)$ for the current behaviour policy π as well as for all states s and actions a. This can be done in essentially the same way as with the state value prediction as shown in the previous section (Sutton and Barto, 1998).

The update rule for TD is given in Equation 3.14, which is performed for every transition from a non-terminal state s_t and if s_{t+1} is terminal then $Q(s_{t+1}, a_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, that comprise the transition from one state-action pair to the next (Sutton and Barto, 1998).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Equation 3.14: Update rule for state-action pairs in TD(0).

The Sarsa algorithm is shown in Figure 3.11. Sarsa's convergence properties are dependent on the choice of action selection used, the guarantee of

infinite visits to all state action pairs and the policy converging in the limit to the greedy policy (Sutton and Barto, 1998).

Initialise *Q(s, a)* arbitrarilyRepeat for each episode:Initialise *s*Choose *a* from *s* using policy d

ε

ε

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$$

Figure 3.11: Sarsa: An on-policy TD control algorithm.

3.3.3.3 Q-learning: Off-Policy Control

Q-learning is an off-policy TD control algorithm suggested by Watkins in 1989, and was an important breakthrough in reinforcement learning (Sutton and Barto, 1998). Equation 3.15 describes it in its simplest single step form.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right]$$

Equation 3.15: Single step Q-learning update rule.

In this definition the learned action-value function, Q, directly approximates the optimal action-value function, Q^*, independently of the current behaviour policy. The behaviour policy affects which state-action pairs are visited and updated. However, in order to achieve convergence to an optimal policy there is the added requirement that all state-action pairs are visited continuously, see Figure 3.12.

Initialise *Q(s, a)* arbitrarily

Repeat for each episode:
Initialise *s*
Repeat for every step in the episode:
Choose *a* from *s* using policy derived from *Q* (e.g., ε-*greedy*)
Take action *a*, observe *r*, *s'*
$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
$s \leftarrow s'$;
until *s* is terminal

Figure 3.12: Q-learning: An off-policy TD control algorithm.

3.3.3.4 Actor-Critic Methods

Generally, the previous methods in this section have operated on the same principles but were altered to cater for different amounts of information and complexity. The *actor-critic* method, however, diverges from the earlier ideas. These are TD methods, which have a separate memory structure to explicitly represent the policy independent of the value function. This policy structure is known as the *actor*, as it selects actions. The estimated value function is still present but is referred to as the *critic* (Sutton and Barto, 1998).

The critic must learn about and critique the current policy being followed by the actor. This critique takes the form of a *TD-error*. This error is the critic's sole output and is the basis of all learning in both the critic and actor. Figure 3.13 shows the architecture of the method (Sutton and Barto, 1998).

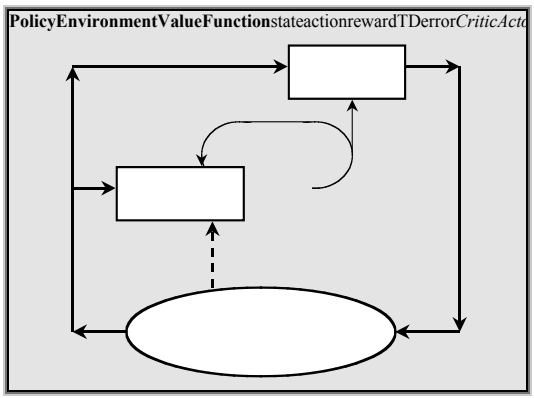


Figure 3.13: The Actor-Critic architecture.

This method is based on the ideas of reinforcement comparison (3.2.3.5). Typically, after each action has been selected the critic evaluates the new state to determine whether things have improved or not (Sutton and Barto, 1998). This is represented by the TD-error calculated using Equation 3.16, where _$_t$ is the TD-error and _ is the discount rate (3.2.2).

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Equation 3.16: TD-error calculation in actor-critic methods

This TD-error is then used to evaluate the action just selected. If the TD-error is positive, then the tendency to select action *a*, should be strengthened and vice versa if the error is negative (Sutton and Barto, 1998). There are two primary advantages to this approach. The first is that it requires minimal computation in order to select actions. The second is that they can learn an explicitly stochastic policy (Sutton and Barto, 1998).

3.4 Advanced Reinforcement Techniques

In the previous section three classes of methods were introduced: dynamic programming, Monte Carlo, and temporal difference learning. All these different approaches solve problems of varying complexity and with different levels of information about the environment (Sutton and Barto, 1998). This section will look at a few techniques that unify the methods above, into a more useful form for the specific problem being addressed in this thesis. Firstly, it will introduce the mechanism of eligibility traces, which smoothly integrates Monte Carlo and temporal difference. Secondly, it will investigate methods for generalizing states and state-action pairs.

3.4.1 Eligibility Traces

Eligibility traces are one the basic mechanisms in reinforcement learning for providing a more general method that may learn more efficiently. They can be combined with nearly all temporal-difference learning methods such as Sarsa and Q-learning. TD(λ) is one such popular algorithm, where λ refers to the eligibility trace. They provide a means of bridging the gap between Monte Carlo methods and Temporal Difference. Therefore, when augmented with TD methods, they produce a collection of methods with Monte Carlo at one end of the spectrum and temporal difference at the other. This is commonly called the *forward* view (Sutton and Barto, 1998).

An alternative view of eligibility traces, referred to as the *backward* view, is that they keep a temporary record of the occurrence of an event, such as an action being taken or visiting a state. Then when a TD-error occurs only those events are assigned credit or blame for the error. Thus, eligibility traces provide a bridge between events and training information. Either view equates to essentially the same basic algorithm (Sutton and Barto, 1998).

This section will first look at the prediction problem as seen by TD(λ). It will then extend this by also looking at predicting action values in order to solve the control problem. More specifically, it will look at the control problem as seen by the actor-critic method. Finally, it will look at a slight improvement to the eligibility trace known as a *replacing trace* (Sutton and Barto, 1998).

3.4.1.1 n-Step TD prediction

Recall that Monte Carlo methods perform a backup for each state, based on the entire sequence of observed rewards from that state to the end of the episode. In contrast, simple TD methods base their backup on only the next step. Clearly, intermediate methods would perform a backup on a number of steps, more than one but less than all of them. These methods are still TD methods because they are relying on estimates being based on later estimates and are called *n-step TD methods*; therefore, simple TD is actually a *one-step TD method* (Sutton and Barto, 1998).

Formally, the *target* in Monte Carlo methods is the complete return calculated using Equation 3.17, where T is the last time step.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \mathrm{L} + \gamma^{T-t-1} r_T$$

Equation 3.17: Calculation of target in Monte Carlo methods.

In *one-step TD methods*, the target is the first reward plus the discounted estimate value of the next state as shown in Equation 3.18.

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$$

Equation 3.18: Calculation of target in one-step TD methods.

Therefore, it follows that the target in *n-step TD methods* consists of all the rewards up to n plus the discounted return of the subsequent state, as can be seen in Equation 3.19. Of course if the episode ends before step n then the conventional complete return is used.

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \mathrm{L} + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

Equation 3.19: Calculation of target in n-step TD methods.

The state value $V_t(s_t)$, the estimate of $V^\pi(s_t)$, can be calculated using an incremental approach as defined in Equation 3.20, where α is the step size parameter. The increment to states other than s_t are $\Delta V_t(s) = 0$.

$$\Delta V_t(s_t) = \alpha\left[R_t^{(n)} - V_t(s_t)\right]$$

Equation 3.20: Increment calculation for n-step state value.

The reason for calculating $V_t(s_t)$ as an increment is to allow for a method known as *on-line updating*, where updates are performed during the episode as soon as the increment is calculated. In this method $V_{t+1}(s) = V_t(s) + \Delta V_t(s)$ for all $s \in S$. While on-line updating does speed learning it still must wait n steps before updating can occur, which complicates implementation (Sutton and Barto, 1998).

3.4.1.2 TD(λ)

In the last section, backups were being done toward an n-step return, but they can also be done toward an *average* of n-step returns. For instance, a backup can be done toward a return that is a half of a two-step return, a quarter of a four-step return and a quarter of an eight-step return, as shown in Equation 3.21. It is in fact possible to average any set of returns in this way provided the weights on the component returns are positive and add to 1. A backup that averages simpler component backups is called a *complex backup* (Sutton and Barto, 1998).

$$R_t^{average} = \frac{1}{2} R_t^{(2)} + \frac{1}{4} R_t^{(4)} + \frac{1}{4} R_t^{(8)}$$

Equation 3.21: Example of averaging n-step backups.

The TD(λ) algorithm is in fact just a special case of averaging *n*-step backups, where each of the weights are proportional to λ^{n-1}, and $0 \leq \lambda \leq 1$. A normalization factor of $1-\lambda$ ensures the weights always add to 1. Therefore, the resulting backup is toward the return R_t^{λ}, called the *λ-return* and is defined in Equation 3.22.

$$R_t^{\lambda} = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

Equation 3.22: The general λ-return function

Basically, the return after one step is given by the weight 1-λ, with the remaining steps all decreasing by λ. If a terminal state is reached, all the subsequent steps have returns equal to R_t. It can be seen in the return function that there are two special cases. The first, is that, if $\lambda = 1$ then the summation term goes to zero and the remaining term reduces to the conventional return R_t, which is the same as the Monte Carlo algorithm. Secondly, when $\lambda = 0$, the *λ-return* reduces to $R_t^{(1)}$, which is equivalent to the one-step TD methods, TD(0).

3.4.1.3 Implementation of TD(λ)

The above forward view however, is impractical to implement because it is *acausal*, using, at each step, knowledge of what will happen in future steps. The backward view of TD(λ) provides a method of approximating the forward view and in off-line cases achieving it exactly (Sutton and Barto, 1998).

It works by adding an additional variable associated with each state, called its *eligibility trace*, denoted $e_t(s)$ for state s at time t. It is initially set to 0, which it will keep until the state s is first visited, when it will be incremented by 1. Then on each time step, the eligibility trace for state s, and all other states with an eligibility trace greater than 0, will decay by $\gamma\lambda$. If state s is subsequently revisited it will again be incremented by 1. Formally, this is shown in Equation 3.23 for all $s \in S$, where γ is the discount rate and λ is the parameter introduced in the last section and is referred to as the *trace-decay parameter* (Sutton and Barto, 1998).

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t. \end{cases}$$

Equation 3.23: Update rule for eligibility trace variable.

Now when we calculate the TD-error this is applied proportionately to each of the states. Therefore, those with a higher *e* value will receive the greatest update, which is calculated using Equation 3.24. The algorithm for the on-line version of the tabular TD(λ) is given in Figure 3.14.

$$\Delta V_t(s) = \alpha\delta e_t(s), \qquad \forall\ s \in S.$$

Equation 3.24: Update formula for a states' value using the backwardview of TD(λ).

Initialise *V(s)* arbitrarily and *e(s)* = *0*, for all $s \in S$

Repeat for each episode:
Initialise *s*
Repeat for every step in the episode:
$a \leftarrow$ action given by π for *s*
Take action *a*, observe *r*, and next state *s'*
$\delta \leftarrow r + \gamma V(s') - V(s)$
$e(s) \leftarrow e(s) + 1$
For all *s*:
$V(s) \leftarrow V(s) + \alpha\delta e(s)$
$E(s) \leftarrow \gamma\lambda e(s)$
$s \leftarrow s'$;
until *s* is terminal

Figure 3.14: On-line tabular TD(λ).

This type of eligibility trace is called an *accumulating trace* because *e* is incremented regardless of its current value and therefore, can continue to increase. Essentially, it is implementing the credit assignment heuristics of *recency* and *frequency*. Thus, an *eligible* state for learning gains more credit when visited recently and more often (Sutton and Barto, 1998).

In some situations a significant improvement in learning has been observed by using a second type of trace called a *replacing trace*, which does not increment *e* by 1 but instead sets it to 1 (Singh and Sutton, 1996). Basically, it can be viewed as disregarding the frequency heuristic, and is compared diagrammatically with the *accumulating trace* in Figure 3.15.

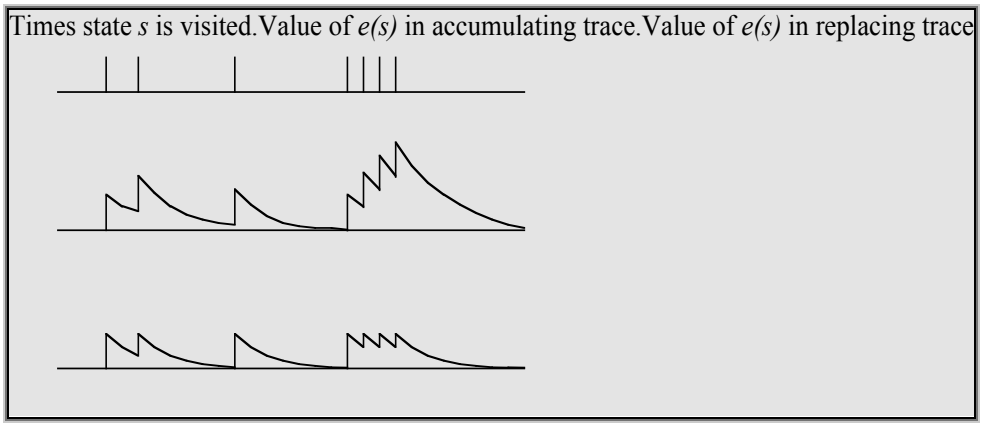


Figure 3.15: Comparison of accumulating and replacing eligibility traces (Singh and Sutton, 1996)

3.4.1.4 Eligibility Traces for Actor-Critic methods

Recall, that the critic in actor-critic methods are state value functions. Therefore, we can simply use TD(λ) for the critic. However, the actor uses state-action pairs and thus requires its own trace variable for each pair. This is implemented simply by altering the single state actor-critic method introduced earlier with Equation 3.25, where $e_t(s, a)$ denotes the trace at time *t* for the state-action pair *s*, *a*.

$$p_{t+1}(s,a) = p_t(s,a) + \alpha\delta_t e_t(s,a)$$

Equation 3.25: Update formula for state-action pairs in the actor-critic method.

3.4.2 Integrating Reinforcement Learning and Neural Networks

During this description of reinforcement learning there has been one fundamental flaw that has not been mentioned. Regardless of the method selected how can we both keep a record of, as well as continually update every state and state-action pair in problems with huge state spaces, such as chess containing roughly 10^{43} positions. With large state spaces many states also may never even be visited once after thousands of episodes, let alone being visited enough to learn a meaningful policy.

Clearly, the only way this can be achieved is through generalization from previously experienced states for the states not yet seen. Fortunately, generalization from examples has been researched extensively for many years across many fields and we need only integrate these methods into our reinforcement-learning model. The type of generalization required is often called a *function approximation* as the examples are from a desired function, such as a value function. Function approximation can be achieved using techniques such as supervised learning neural networks as described in chapter 2, pattern recognition and statistical curve fitting (Sutton and Barto, 1998).

3.4.2.1 Value Prediction with Neural Networks

Generally, the approximate value function V_t at time t has been stored for every state, probably in a table. However, when building a generalized function we instead represent this data in a parameterized functional form with a parameter vector $\vec{\theta}_t$. Therefore, the function V_t now depends totally on $\vec{\theta}_t$ and varies from time step to time step only as $\vec{\theta}_t$ varies. This is particularly useful when implemented with a neural network, as they are primarily a means of calculating vector equations (Sutton and Barto, 1998).

For instance, if V_t is the function computed by the network with $\vec{\theta}_t$ being the vector of connection weights, then by adjusting these weights any number of a wide range of different functions V_t can be implemented by the network. Generally, $\vec{\theta}_t$ does not have to have a value for every state. Instead each value can represent a number of similar states and therefore, when that value

is adjusted a number of states have their value functions changed (Sutton and Barto, 1998).

The problem here is that neural networks using supervised learning are expecting training examples that they can attempt to converge towards. Therefore, the return R_t from the environment must be used as the error value for the network to use in adjusting weights through backpropagation. Thus, we must view each backup as a conventional training example (Sutton and Barto, 1998).

When using supervised learning we are generally seeking to minimize the mean-squared error (MSE) over some distribution, P, of the inputs. In the value prediction problem the inputs are states and the target function is the true value function V^{π}, therefore, the MSE for an approximation V_t, using $\vec{\theta}_t$, is given in Equation 3.26. The distribution P is important, as it is not usually possible to reduce the error to zero for all states (Sutton and Barto, 1998). Therefore, P acts as a guide on how these function approximations can be balanced.

$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s)\left[V^{\pi}(s) - V_t(s)\right]^2$$

Equation 3.26: Calculation of MSE for an approximation V_t using $\vec{\theta}_t$.

Basically, integration of TD(λ) and neural networks is achieved by taking the TD-error, δ, as defined earlier, and adjusting our network's vector of weights by applying Equation 3.27, where $\vec{e}_t$ is a vector of eligibility traces.

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t$$

Equation 3.27: Vector weight update rule.

There is one element in $\vec{e}_t$ for each element in $\vec{\theta}_t$ and they are updated using Equation 3.28. The complete algorithm for the gradient-descent TD(λ) is given in Figure 3.16.

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t)$$

Equation 3.28: Eligibility update rule

Initialise $\vec{\theta}$ **arbitrarily and** $\vec{e} = \vec{0}$

Repeat for each episode:
s initial state of episode
Repeat for every step in the episode:
$a \leftarrow$ action given by π for *s*
Take action *a*, observe *r*, and next state *s'*
$\delta \leftarrow r + \gamma V(s') - V(s)$
$\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
$\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
$s \leftarrow s'$;
until *s* is terminal

Figure 3.16: On-line gradient-descent TD(λ) algorithm for estimating V^{π}.

3.5 Conclusion

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision-making. It, like many artificial intelligence techniques, stems from the psychology of animal learning but is distinguished by its emphasis on learning by the individual agent from direct interaction with the environment. It does this without the need for supervision or a complete model of the environment. It achieves this through the use of a formal framework defining the interaction between a learning agent and its environment in terms of states, actions and rewards. This framework represents the essential elements involved in learning, such as a sense of cause and effect, a sense of uncertainty and non-determinism, and the existence of explicit goals.

This chapter has described the reinforcement-learning model in detail, describing all of the primary elements and how they interact. It then described a number of methods used to evaluate their actions and explore a problem domain through careful action selection. It has also given a detailed description, including procedural algorithms and equations for the three main categories of methods used in reinforcement learning: Dynamic Programming, Monte Carlo and Temporal Difference. Finally, it discussed a technique, called eligibility traces, for amalgamating the Temporal Difference technique smoothly with Monte Carlo methods. This allows for on-line learning and is the foundation for the TD(λ) algorithms used in this thesis.

CHAPTER 4 Computers Playing Games

Chess is in its essence a game, in its form an art,
and in its execution a science.

Baron Tassilo

Games have engaged the intellectual faculties of people for as long as civilization itself. The popularity of board games such as chess and Go are in part because they offer pure, abstract competition and this results in an "…idealization of worlds in which hostile agents act so as to diminish one's well being" (Russell and Norvig, p122, 1995). This also makes them an appealing target for AI research (Russell and Norvig, 1995). In addition, the state of a game is easy to represent, all necessary information is provided and the agent is restricted to selecting from a relatively narrow set of possible actions.

Until recently, a lot of research in game playing has relied primarily on brute force searching approaches rather than any formal AI method. This is mainly because these brute force methods have provided a means of playing at a reasonable competence and with the development of Deep Blue (4.3.1) the ability to compete against the best human players in the world (Schaeffer, 2000). However, these methods may not be able to exceed human ability, except with much deeper searching, as they need human expert knowledge in the form of *open-books*, to perform as well as they do. The application of AI techniques could allow a computer to learn how to play for itself. This would provide the potential, theoretically, for the computer to exceed human expertise and provides a more interesting challenge.

This chapter will first provide a brief history of machines playing chess before describing some of the primary methods used in the leading *brute force* chess and game playing programs. It will then provide a number of case studies of game playing programs, using both brute force and reinforcement learning based approaches.

4.1 History of Chess Playing Machines

People have been fascinated with being able to make a machine play chess. For instance, Baron Wolfgang von Kempelen's mechanized chess playing machine, called *The Automaton Chess Player*, which was first exhibited by him at the Royal Palace in Vienna in 1770. This machine played very good chess and nearly always won but had a man elaborately hidden inside. While it was strongly believed to be a hoax, it was not revealed until 1824 (Levy, 1976).

The first genuine work on mechanical chess was in Charles Babbage's paper *Passages on the Life of a Philosopher* in 1864, where he considered how his analytical engine might play the game and even formulated some simple rules such as lookahead (Smith, 1999). At around 1900 Torres Quevedo constructed an actual machine that played the endgame of king and rook against king, which worked well providing the original setup conformed to a particular pattern (Hayes and Levy, 1976).

Alan Turing and D.G. Champernowne carried out the first work in computer chess in 1948. They developed a one-move analyzer called the TUROCHAMP. At the same time D. Michie and S. Wylie also created a one-move analyzer. In 1950 Claude Shannon published a paper on computer chess that remained the most significant contribution to computer chess for the next two decades (Hayes and Levy, 1976). In this paper he described two possible strategies for computer chess (and deterministic game playing in general). The first, "type-A", was essentially a brute force approach consisting of a fullwidth minimax search with static evaluation. The second, "type-B", involved searching particular lines more deeply than others, sometimes called plausibility analysis and also introduced the concept of *quiescence* (4.2.3.4) (Smith, 1999).

Most developments that have tried to play chess well have used these search techniques applying some means of representing each board position into a numerical value, representing the virtues or dangers of that position. However, some work has been done that experimented with other techniques. For instance, Newell and Simon developed a chess program that used the concept of goals

that represented some feature of the chess situation, such as king safety, material balance, and centre control (Hayes and Levy, 1976).

4.2 The Basics of the Computer Chess Algorithms

The general method used for playing chess in computers is to first generate all the possible legal moves from the current position and then evaluate each of these to find which is best and perform that move. Of course, we ideally want to look further ahead than just one move. Therefore, we generate every move that the opposition player could make in reply to each of our possible moves and evaluate the positions, giving us a better idea of which move to take. This process is effectively looking two half moves ahead and is called *2 ply*. Thus, a higher number of ply used to generate a move should result in a better selection.

The problem here is that we will run out of computer memory and use up too much processor time before we can look far enough ahead, for example there are 169,518,829,100,544,000 possible board positions after only the first 10 moves in the game of chess (anon, 2001). This section will describe the basic methods used in modern game playing programs to avoid this explosion of states. First though, it will detail what methods are generally used for evaluating board positions.

4.2.1 Board Evaluation

Board evaluation, or the evaluation function, is an estimate of the expected utility of the game from that position. Basically, this function uses various features of the board position and gives each a numerical value. A number of features from the boards state can be extracted some of the well used ones are detailed below.

Material value:

Each piece is given a weighting. For instance, in chess each pawn could be given a value of 1, a knight or bishop 3, a rook 5, and a queen 9. Therefore, the total material weight for each player is simple added for each position and the move leading to the position with the highest weight advantage is taken.

Mobility:

Mobility works on the idea that the more possible moves a piece has to work with the more effective it is. For instance, with material value alone two knights and a bishop would be valued as the same as a Queen, but they have much more mobility and can have more effect on the game. Mobility is usually calculated by finding which position has the most possible moves, and identifies that as being the best option. Interestingly, the most favored opening move by grandmasters is P-K4, which also provides the greatest mobility for subsequent moves of any of the 20 possible opening moves (Levy, 1996).

Control:

More subtle methods of evaluation are often used but can be progressively harder to implement. One often used involves placing a value on different areas of the board in relation to various pieces at particular times. For instance, it is often acknowledged that the centre of the board is a powerful position to control during the start and middle stages of the game (Levy, 1976).

Most programs implement at least the first two. Studies have been done showing that control can often be achieved by a well-selected mobility rule (Levy, 1976). Apart from simply selecting which features to use, the developers must also find a numerical weighting that expresses the importance of one feature relative to another. For instance, it is not advisable for the computer to sacrifice its pawns at the start of the game in order to improve mobility (Op cit).

To succeed in implementing an effective evaluation function the development team must be able to access significant amounts of expert knowledge: both to extract the ‘best’ features and to find a reasonable weighting between those features. Essentially, it is this evaluation function that any intelligent agent would be required to learn in order to play such a game and is the core function being learnt in this project.

4.2.2 Minimax search

The minimax algorithm is the basic search tool used in most game programs. The algorithm gives each player a name, the first player, the computer, is called *Max* as it is trying to gain the greatest return for itself. The opposition is called *Min*, and is trying to minimize the computers return. Hence, the algorithms name (Schaeffer, 2000).

Minimax works by building a depth-first, left-to-right, search tree with alternating moves by Max and Min. Each leaf node in the tree is assigned a value, or *score*, computed using an evaluation function as described above. If the leaf node is actually a terminating position then it is given a value representing a win, lose, or draw. This value is then passed back up the tree to the root node. The value selected by a node is the maximum of all its children if it is a Max node or the minimum if it is a Min node (Schaeffer, 2000).

Figure 4.1 shows an example of a tree using minimax search. Max nodes are represented with a square, while Min nodes have a circle. The number shown at each node is the best value achievable for that node from all of the children and is used by the parent to select which branch to follow. The dark line shows which is the best path to be taken by the computer, or Max.

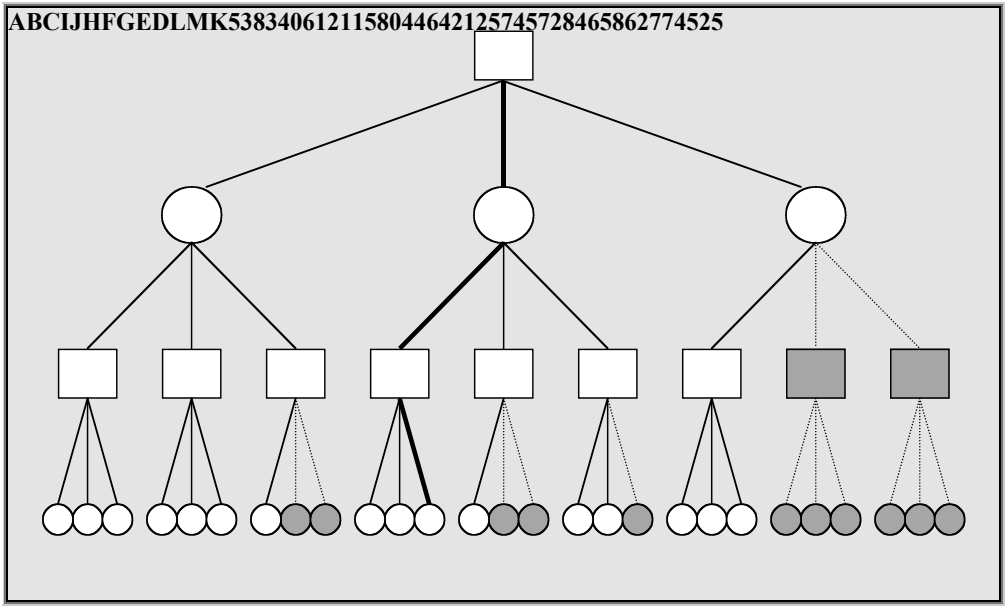


Figure 4.1: Searching a minimax tree (Schaeffer, 2000).

4.2.3 Alpha-beta Pruning

The primary problem with minimax search is clearly its exponential growth as it must examine $O(w^d)$ leaf nodes, where w is the width when we assume a fixed number of branches and d is depth or the number of ply (Schaeffer, 2000). A major improvement that came about in the late fifties is called alpha-beta pruning. Basically, this is a method of cutting off branches that are clearly not going to affect our eventual choice.

If we look again at Figure 4.1 the dark grey nodes have been cut using alpha-beta pruning. For example, when performing our search from left to right and node G is reached we perform an evaluation of the first child and find that its value is 6. Due to node G being a maximizing node, we know our eventual value at this node will be six or greater regardless of the value of the remaining children. We also know that node B will select the minimum out of its child nodes: E, F, and G. Now, as node F with value 4 is less than 6, we know that regardless of the return from the remainder of Gs children, that node F will still be selected. Thus, we cut the remaining children and do not evaluate their returns.

The algorithm searches the tree by using two parameters: α and β. α is the minimum value that player Max has achieved, while β, is the maximum value to which player Min can limit Max to. Then if at any node where the return results in the condition $\alpha \geq \beta$ causes a cut off to occur (Schaeffer, 2000). This enhancement to the minimax search improves the best case behaviour of the search to $O(w^{d/2})$. This best case occurs when the move leading to the best minimax score is searched first at each interior node during traversal. The worst case occurs when the best move is searched last, in which it results in a standard minimax search of $O(w^d)$(Op cit).

One problem with the alpha-beta pruning enhancement is that if there is a minor error in the evaluation function it can remain hidden for long periods of time. This is due to the error only materializing when it has propagated to the root, which in a deep and wide search may occur very little. Therefore, some game playing programs have had bugs for years before the right sequence of events occur that cause it to be revealed (Schaeffer, 2000).

There are also a number of additional enhancements that can be made that further improve performance. These methods have been grouped into four types and are discussed in the following sub sections. A search can incorporate any or all of these methods depending on the specifics of the game being played and personal preference.

4.2.3.1 Caching Information

One characteristic that is common in many games is that two different sequences of moves produce the same result. Therefore, the sequences are transposable into each other. This means that the concept of a search *tree* is a misnomer; it is really a search graph. Caching information methods try to identify these sequences and eliminate redundant branches (Schaeffer, 2000).

This is generally accomplished by building a *transposition table*, usually a large hash table, of recently searched positions. Then before a node is searched the table is queried to find if it has been previously searched and if so this information is used instead. The effectiveness of this method varies a lot according to the game. For instance, it can achieve approximately a 75% reduction in chess but only 33% in games like Othello (Schaeffer, 2000).

4.2.3.2 Move Ordering

The difference between the search-tree size in the best and worst cases using alpha-beta pruning hinges on the order in which moves are considered. Therefore, it makes sense to order them so that the most promising moves are searched first. This is especially important at the root when a cutoff saves searching a large area of the tree (Schaeffer, 2000).

Most alpha-beta-based game programs implement this method using a technique called *iterative deepening*. This works by first doing a 1 ply search and then re-ordering the moves according to the results. The tree is then searched again to a depth of 2 ply, and so on. The idea is that the best move for a *(d-1)*-ply search is likely to also be best for a *d*-ply search (Schaeffer, 2000).

Clearly, this can result in some repeated searching during the earlier ply but the chances of an improved search later on means that the cost is worth it. Generally, it results in searching the best move first in over 90% of the time in chess and checkers and over 80% of the time in Othello (Schaeffer, 2000).

4.2.3.3 Search Window

The alpha-beta algorithm searches the tree with an initial search window of $[-\infty, +\infty]$. However, usually the extreme values do not occur and there will be a move somewhere that will fit into a narrower range. Therefore if we start with a smaller window we can prune branches at the start of our search before we find our best path (Schaeffer, 2000).

One method, called *aspiration search*, centers the window on the expected value and sets the window to be plus or minus a reasonable range (δ). It can also be combined with iterative deepening by using the (d-1)-ply result as the center. The value of δ is application dependent and is determined by empirical evidence (Schaeffer, 2000).

4.2.3.4 Search Depth

Sometimes, when arbitrarily breaking off a search at a particular predefined depth, situations can occur where the evaluation function gives an incorrect score. For example, if in the last move leading to the leaf node the black Queen takes a white knight, the evaluation function would then score the board as black having a knight advantage. However, if the next ply had been considered then it may have been observed that the black queen was then capture by white, which is a far worst outcome for black (Scott, 2001).

This problem can be resolved by varing the depth of the search according to the likelihood of it being a winning branch. This allows the program to gather more information about potentially good moves. This tends to be an application specific means of improving a search (Schaeffer, 2000). For instance, a chess program may extend a search where a checking move or a capture was found, as this usually indicates that something interesting is happening.

One popular technique that can be applied to many games is a search technique called *quiescence search*. This technique attempts to ensure that a leaf node being evaluated is stable or *quiescent*. Therefore, if the node has just had a capture or threat applied a small search is performed beyond that node to resolve the eventual outcome of the event. Only when a stable node is found is the evaluation function applied (Scott, 2001).

Another popular technique is called *singular extensions*. These are applied when a *forced* move is found. This is achieved by manipulating the search window to see if the best move is significantly better than the second best move. When such a move is found, that line of play is extended. The idea is that a forcing move indicates an interesting property of the position that requires further exploration (Schaeffer, 2000).

4.3 Case Studies

The majority of game-playing programs use the above techniques, or something similar to determine their moves, ultimately leading to programs today that can sometimes defeat human champions. However, there have been attempts to develop methods of learning evaluation functions and search techniques over many game types. This section is going to first review the most famous chess computer Deep Blue as an example of the above brute force methods. It will then investigate some learning methods that have had varying degrees of success. These also offer an insight into approaches of applying reinforcement learning techniques to game playing.

4.3.1 Deep Blue

Deep Blue was the first chess computer to defeat a reigning world chess champion, Garry Kasparov, in a regulation match (Campbell, 1999). It was developed by a team of people funded by IBM and the principal scientists who developed the program were Feng-hsiung Hsu, Murray Campbell, and Joe Hoane. Deep Blue used a specifically designed VLSI chipset that incorporates an alpha-beta search engine and is capable of analyzing over two million chess positions per second. It also uses iterative deepening and transposition tables and was the pioneer of singular extensions (Schaeffer, 2000).

In addition to brute force search Deep Blue also uses an extended open book database of over 700,000 Grandmaster chess games. When a board position is found that is in the database it formulates an evaluation based on who played it and how favorably the game ended. These games receive a positive or negative number, representing a bias that influences the standard evaluation of that particular line of play. Finally, Deep Blue had an extensive database of endgame table bases for all 5 piece, or fewer endgames, allowing it to play perfect play during game endings.

4.3.2 Samuel's Checkers Program

Arthur Samuel developed a checkers program in the early 1950s that used the first real learning algorithm for a full board game. He was not interested in developing a program that could play checkers well; rather, he was interested in creating a program that *learnt* how to play checkers (Samuel, 1959). The program, however, could play very well defeating Robert Nealey in a 1963 exhibition match. What is interesting about his algorithm is that it uses what would be referred to today as temporal difference learning (Sutton and Barto, 1998).

Samuel's program played, by performing a lookahead search from the current position. Essentially, it performed a minimax search using varying depths and branch cutoffs that were analogous to alpha-beta pruning. Samuel's program also implemented two main learning methods. The first, which he refers to as *rote-learning* consists simply of storing a description of each board position encountered during play together with its *backed-up* value, or evaluation score, determined by the minimax procedure. Then, when performing the minimax search in later games, if an identical board position was encountered it used the board evaluation of the saved position, effectively amplifying the effectiveness of the search, see Figure 2.2(Samuel, 1959).

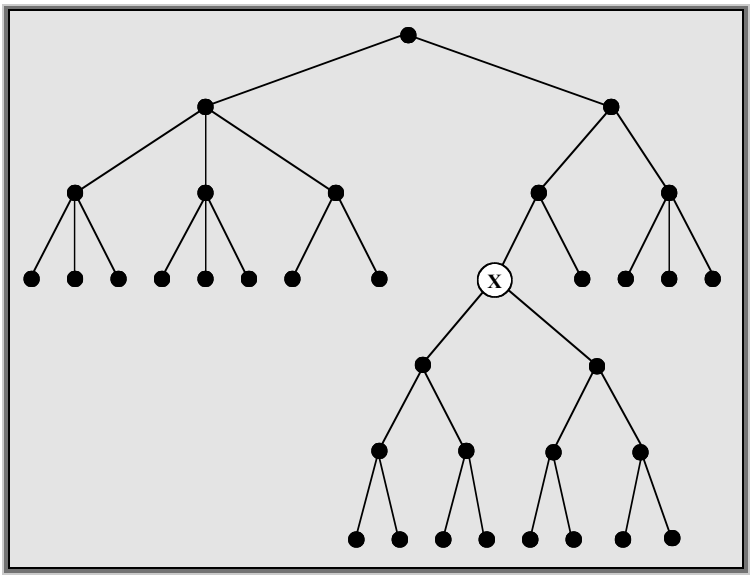


Figure 4.2: Simplified representation of the rote learning process, where previously learnt information effectively increases the ply of the backed-up score in Samuel's checkers program (Samuel, p213, 1959).

In order to avoid the program from choosing the longest of a group of paths that would yield the same result, each saved board position was given a weighted representation by decreasing the position's value a small amount with each backup. Then during minimax analysis, if two board positions differed by only there ply number then the one with the lowest ply was selected. This introduced a 'sense of direction' into the programs play (Samuel, 1959).

The second form of learning, called *learning by generalization*, involved a lengthy procedure for the program to modify the parameters used in the evaluation function (Samuel, 1959). The central concept he used here was similar to Tesauro's method used in his TD-Gammon (4.3.3)(Sutton and Barto, 1998).

Essentially the program played many games against an older version of itself, called alpha and beta respectively. The alpha version had a set of parameters that were untried while the beta version had the best evaluation function currently found. At the end of a game if alpha is judged to have played better then it gives it evaluation function to beta to use in subsequent games (Samuel, 1959).

4.3.3 TD-Gammon

One of the most impressive examples of reinforcement learning is Gerry Tesauro's backgammon playing program, called TD-Gammon. This program was given very little knowledge about the game and how to play but managed to learn purely through self-play to a standard that was equivalent to the best computer playing programs of the time (Sutton and Barto, 1998).

Backgammon is a game involving good strategy and an element of chance. The game does not lend itself to the standard heuristic search techniques because it can have a branching factor of about 400, compared to chess with an average of 35. TD-Gammon 0.0 uses a nonlinear form of TD(λ), where the estimated value, $V_t(s)$, of any state s was an estimate of the probability of winning starting from that state (Sutton and Barto, 1998).

To do this, rewards were defined as zero for all time steps except those that win the game. The value function was implemented using a standard multilayer neural network. The network consisted of three layers: the input layer had 198 input neurons to represent the board, the hidden layer which had 40 hidden units and a final output layer with one output unit. The output gave a single value representing the estimated probability of winning from that position (Sutton and Barto, 1998).

The network learnt by using the TD-error to adjust connection weights between neurons using the gradient descent method with the gradients computed by the error backpropagation algorithm described in chapter 2. After playing approximately 300,000 games against itself, TD-Gammon 0.0 learned to play approximately as well as the best previous backgammon computer programs. It was then combined with Neurogammon, a program developed earlier by Tesauro using significant expert knowledge as training examples. The combination resulted in TD-Gammon 1.0, which performed substantially better than all previous backgammon programs. Versions 2.0, 2.1 and 3.0 introduced short selective searches of 2 and 3 ply improving performance up to what is regarded today as a program that is as good as the best human players in the world (Sutton and Barto, 1998).

One interesting development that came out of the way TD-Gammon played was how it affected human play. For instance, during its training process it learned certain opening positions differently to what was convention amongst the best human players. Based on the success of TD-Gammon and from further analysis of these opening moves the human players now play these positions as TD-Gammon does (Tesauro, 1995). Thus, by not initially using expert knowledge it has shown that the technique can develop methods that are not only different but improve human understand of the problem domain.

4.3.4 NeuroChess

Popularity in reinforcement learning techniques was generated, due to the success of Tesauro's TD-Gammon collection of programs causing attempts to build game playing programs using TD(λ). NeuroChess, developed in 1995 by Sebastian Thrun, attempted to use very similar techniques as those used by TD-Gammon. Additionally, it also used a neural network form of explanation-based learning, which analyzed games in terms of a previously learned neural network chess model (Thrun, 1995).

NeuroChess employs the basic variant of temporal difference, TD(0), and ranks board positions according to their 'goodness'. It does this by transforming entire chess games, denoted by a set of chessboard *S*, into training patterns for the evaluation function, *V*. The learning rule assigns, the terminal board position, a 1 for a win, a –1 for a loss and 0 for a draw. The rewards assigned for the intermediate chess boards, those leading up to the final position, are given the value calculated using Equation 4.1, where γ is the constant discount factor set at a value of 0.98 (Thrun, 1995).

$$V^{target}(s_t) = \gamma \cdot V(s_{t+2})$$

Equation 4.1: Recursive update rule for intermediate board possitions used in NeuroChess (Thrun, 1995).

Due to the complexity of chess and the occurrences of special cases such as *knight forks*, Thrun included *explanation-based neural network learning* (EBNN) techniques to help guide the learning of the TD(0) network. The EBNN contains domain specific knowledge represented using a separate neural network, called the *chess model*, denoted *M*. *M* was previously trained using a large database of grand-master chess games. Once trained it was used to map arbitrary chessboards s_t to the corresponding expected board s_{t+2} two half moves later. The EBNN then exploits *M* to bias the board evaluations generated by the TD(0) network (Thrun, 1995).

NeuroChess has shown that it is able to defeat GNU-Chess, a publicly available chess tool, which is frequently used as a benchmark for chess programs. However, generally its play compares poorly to both GNU-Chess

and human players. According to Thrun it has two fundamental problems. The first was that training time was limited and excessive training is required for better play. The second problem is that with each step of TD-learning NeuroChess losses information. This is partially because board descriptions used are incomplete, for instance "…knowledge about the feature values alone does not suffice to determine the actual board exactly"(Thrun, p1075, 1995). Most importantly though is that neural networks can not assign arbitrary values to all possible feature combinations (Thrun, 1995). Thrun concludes that it is unclear whether a TD-like approach as he used is capable of playing good chess.

4.4 Conclusion

While Samuel's checkers program introduced a temporal difference learning like technique in the 1950s it was overshadowed by the search based algorithms, primarily because these achieved greater improvement as computers improved. It wasn't really until Tesauro revisited the technique, with his TD-Gammon program, that temporal difference learning, within the domain of game playing, gained much interest. However, this enthusiasm for the algorithm, despite several attempts, has not as yet materialized into a repeated performance for other board games such as Othello, Go or chess. NeuroChess was one such attempt, which did have some success but did not meet the expectations placed on it.

Many authors have discussed what the peculiarities of backgammon are that make it suitable to TD(λ). Mainly, it is believed that the smoothness of the evaluation function and the game's stochasticity are important factors. Another reason is believed to be that human players may only look a couple of moves ahead because of the difficulty of knowing what will happen due to its randomness. Therefore, it is competing in a "…pool of shallow searches" (Baxter et al, p246, 2001). In the game of Chess however it is difficult to develop a reliable tactical evaluation function without also using a deep look-ahead search. A search of great depth though requires a simple evaluation function to ensure faster scoring of the board positions and this seems to rule out

"…the use of expansive evaluation functions such as neural networks" (Baxter et al, p 246, 2001).

This Chapter has provided a brief overview of the history and algorithms used in chess and game playing programs. It then discussed four case studies to show both the well-established search techniques as applied in Deep Blue and the new attempts to learn game playing. Generally, it has not been shown yet whether TD(λ) can or cannot play the general game, but so far its application to deterministic games have not fully succeeded.

CHAPTER 5 Methodology

It is the aim of the modern school, not to treat every position according to one general law, but according to the principle inherent in the position.

Richard Reti

During the previous three chapters a number of AI techniques and game specific methods were discussed. Primarily, an extensive investigation of reinforcement learning algorithms was done. This served to both illustrate how the methods have evolved and show which methods are available to use in this project. An overview of the standard algorithms used by the majority of game programs available today was also studied.

Recalling from the introduction, the aim of this thesis is to investigate whether reinforcement-learning techniques are capable of learning to play selected chess end games. Clearly, due to the large number of techniques available, as seen in chapter three, not all the various methods can be tested. Instead, the most likely algorithm to succeed, TD(_), was selected, as this was used by Tesauro in his successful TD-Gammon program and is best suited to this environment. In addition a number of variations have been investigated, taking the form of separate implementations, each with their own agent that use different information about the game or have different variable settings.

The implementation of the agents involved a number of design decisions, which this chapter will detail. First an outline of the general algorithms applied and the program's basic design is described. Then, each of the main components and their main methods will be explained in detail. The individual agents' state-views and parameter settings are also described along with predictions of what, if anything, they are expected to learn. Finally, an outline of the method used to train the various agents is provided.

5.1 Algorithm Overview

In the last chapter the basic game algorithm that was discussed identified three primary components: board evaluation, look-ahead search and an open book utilising expert knowledge. The main element of concern to this thesis is the board evaluation function. Traditionally, programmers and experienced chess players craft this function from their knowledge of the game. As this project's aim was to build an application that learns how to play chess for itself it is this function that is the key.

In keeping with previous work in reinforcement learning as applied to board games the underlying algorithm employed in this thesis was TD(λ) where λ has the value of 0. Recall that the λ value refers to the eligibility trace and that when λ = 0, the λ-return reduces to $R_t^{(1)}$, which is equivalent to the one-step TD methods. Therefore, the method actually implemented is most closely related to the actor-critic method (3.3.3.4).

The actor-critic method contains two separate memory structures. The first is the actor, which selects which action to be taken by the agent. The second is the critic, whose task it is to learn about and critique the current policy being implemented by the actor. Basically, The actor selects which move to take by analysing each board using the agents' neural network. The next time the agent has a turn it again selects its best move. The critic then compares this chosen position with the earlier one and calculates the TD-error. It then uses this error information to adjust the actor's policy so that the previous state reflects this new information about its eventual position.

In supervised learning the neural network is trained through the use of a teacher. This teacher is in the form of a set of training data, where the correct output expected from the network is known. Therefore, when it gets the wrong answer it is corrected by feeding the right result back into the network. However, in TD, there is no teacher. Instead, there is only the critic, who does not know the correct result. All the critic knows is whether the agents' general position is improving or worsening. The critic also uses the same neural network used by the actor to make this judgement and therefore, is learning alongside the actor.

5.2 Design

The basic design used for implementing the core TD agent in this thesis is shown in Figure 5.1. The program has been broken into three primary classes: the GameEngine representing the external environment, the TD-Agent and its associated neural network. These components are described briefly below and in more detail in the following three sections.

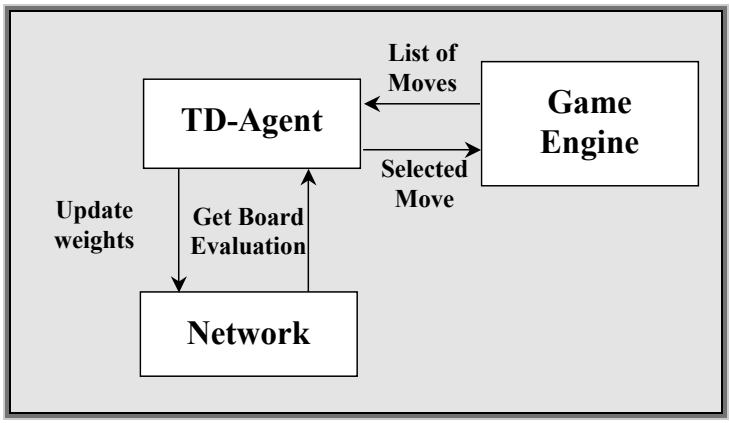


Figure 5.1: Basic program design used for agent training and testing.

The GameEngine component handles all the chess specific code. It is given the current board position and information about whose turn it currently is. Using this information it can generate a single ply of legal moves in the form of new board positions and test whether the game is complete. The class also provides some helper functions that can test if a square is under attack, or if the piece in that square is attacking or defending another square.

All of the legal board positions are then passed to the TDAgent class. This class assigns a value for each board position by first converting the board into a format suitable for the agent's network. This forms the networks input, which produces a corresponding output value that is assigned to that board position. The agent then uses these values to decide which board is the most desirable and selects the move that led to that board position. Finally, when the agent gets the next group of board positions for the following turn, it finds the value difference between this turns starting position and last turns, plus any reward assigned from a completed game, to train the network.

5.3 Game Engine Implementation

This class acts as the outside environment to the agent. It controls the overall game of chess, including move generation and testing for terminal states. This section details the implementation of these elements.

5.3.1 Board Representation

In implementing the game engine the first stage was to define the board representation. The depiction selected, was to use a simple 8x8 array with a number identifying which piece if any is on each square. Positive numbers were used for white and negative for black pieces. The square was assigned the value 0 if no pieces were present.

```
int boardSqrs[8][8] =   {  {-2,  -3,  -4,  -5,  -6,  -4,  -3,  -2},
                           {-1,  -1,  -1,  -1,  -1,  -1,  -1,  -1},
                           { 0,   0,   0,   0,   0,   0,   0,   0},
                           { 0,   0,   0,   0,   0,   0,   0,   0},
                           { 0,   0,   0,   0,   0,   0,   0,   0},
                           { 0,   0,   0,   0,   0,   0,   0,   0},
                           { 1,   1,   1,   1,   1,   1,   1,   1},
                           { 2,   3,   4,   5,   6,   4,   3,   2}};
```

Figure 5.2: Board representation of starting position for full game of chess.

Figure 5.2 shows an example of a full game of chess starting position using the chosen representation. Table 5.1 shows what numbers correlate with which type of piece.

Number		Piece Type
White	Black	
0		Empty Square
1	-1	Pawn
2	-2	Rook
3	-3	Knight
4	-4	Bishop
5	-5	Queen
6	-6	King

Table 5.1: Piece numbering used in board representation.

5.3.2 The Game Loop

The Game engine is responsible for the core game control, which is illustrated in Figure 5.3. This code simply loops once for every move until the game is complete. For each move the next level of legal moves are generated and passed to the agent. The agent selects its next move and updates its network.

```
//      Setup starting board and place in a node called root.

NumMoves   = 0;
turn       = 1;    //      Set  the white player to go first

//      Keep looping until a terminal state is found or until arbitrary cut-off point.
//      The cut-off is needed as chess is not guaranteed to ever reach a terminal state.
while (( stillPlaying( root, num, turn ) ) && ( numMoves < 5000 )){
      num = generateMoves(root, turn);            // Generate legal moves.
      root = agent->selectMove(root, num, turn); // Get agent to select a move.
      agent->updateNetwork(root, 0.0, turn);      // Calc TD-error and update.
      turn = turn * -1;                           // Change players turn
      numMoves++;                                 // Increment number of moves.
}
```

Figure 5.3: Main game loop.

It can be seen here that the agent selects the move for both players. This means there is no outside influence on how the game is played and therefore no external expert knowledge is introduced into the system. This, however, introduced an unexpected problem. Due to the agent controlling both players it can get stuck in a sequence of repeating moves and therefore never reach a terminating state.

This is one of the major drawbacks of chess over most other board games, in that it is not guaranteed to ever terminate. When humans play they can recognise this situation and agree on a draw. The computer, however, cannot identify it easily, especially if there are many different moves in the sequence. Two solutions for this have been implemented to resolve this problem. The first was to introduce an arbitrary cut-off at 5000 moves. Through testing it was found that most games reaching this number of moves were trapped in a sequence. To implement this process of cutting a game short the agent cannot be allowed to alter its weights during such a game as this could distort its learning process. Therefore, the agent saves the networks' weights at the end of each completed game and reverts back to those weights when a game is cut-off for running to long. The second is handled by the agent itself and is discussed in section 5.4.1.

5.3.3 Move Generation

A function called `generateMoves` was written to produce the next level of valid moves given the current position. The function takes two parameters: the first is a pointer to a `Node` containing the current board, referred to as the root. The second parameter is an `int` and refers to the current players turn, 1 for white and –1 for black. As each legal move is found the resulting board position is added to the tree. Finally, the function returns the number of moves found.

Moves are found by first stepping through every square on the board. When a piece is found of the correct colour all its valid moves are then calculated. This is achieved by traversing a set of arrays that contain all the possible moves for that type of piece. Each resulting position is then tested for legality. For a move to be valid it must meet a number of criteria. Namely, that it is still on the board, that the square is unoccupied or held by an opposition piece, and its own king must not be under attack by the opposition after the move is complete.

There are a number of special moves and pieces that cannot be fully resolved using the above method. Firstly, a pawn cannot take a piece when travelling in a forward direction, only diagonally. Nor can it travel diagonally if it is not capturing an opposition piece. A pawn is also able to move two squares forward on its first move and when it does so opens itself up to an attack called ‘en passant’ (1.4). Finally, when a pawn advances to the final row it can be upgraded to another piece. For simplicity, in this thesis it is assumed that it should be upgraded to a queen.

A second group of special moves, called castling. There can be either a kings-castle or a Queens-castle and each player can only castle once in a game. All of these special moves are handled through specifically written code for each. Also, due to each of these special moves only being allowed at particular times a number of flags are also stored with every board indicating which are allowed in subsequent turns. Therefore, a board is defined in the form shown in Figure 5.4.

```
struct Board{
    int square[8][8];       //  The 64 squares on the board.
    char* lastMove;         //  String identifying the move taken to reach this board position
    bool canDoEnPassant;    //  True if the opposition did a double pawn advance last turn.
    bool BKCastle;          //  True if black can do a Kings castle.
    bool BQCastle;          //  True if black can do a Queens castle.
    bool WKCastle;          //  True if white can do a Kings castle.
    bool WQCastle;          //  True if white can do a Queens castle.
    double value;           //  The boards value as calculated by a board evaluation function.
};
```

Figure 5.4: Data structure used to represent a board position.

This structure also contains two other variables. The first is a char pointer called lastMove, which contains a string representing the move taken to reach that particular board position. The string uses a modified form of the algebraic notation (Appendix A) commonly used in competition chess. The modification is that the string also identifies the square the piece comes from (except during castling moves). This was included primarily for tracing games during debugging to ensure the agent was playing correctly. The second additional variable was the double called value and is discussed in section 5.5.

5.3.4 Game Termination

After each turn the board is checked to see if it is a terminal state by calling the function `stillPlaying`. This function first checks to see if there are pieces other than the two kings on the board. This is because if there are only kings on the board it is a declared a draw. Secondly, if the number of moves available to the current player is 0 then it must also be a terminal state.

The function then tests to see if either of the kings are under attack, thereby declaring the player doing the attacking the winner. Alternatively, if no king is under attack then the game is declared a draw due to a stalemate. If the function found the board to be a terminal state position then it informs the agent of the result. It then returns false so that the game loop can restart a new game.

5.4 Agent Implementation

The agent has been written using the TD(_) algorithm with a _ value of zero. While, a higher _ value could increase learning speeds, this thesis used zero for both its simplicity and because most work in this field, such as Sebastian Thrun's NeuroChess and Gerald Tesauro's TD-Gammon used this _ value. Therefore, the agent's implementation is relatively simple. It has two primary tasks. The first is to evaluate all the board positions available and to select a move to be performed. Secondly, it must calculate the TD-error, including any reward and update the network appropriately.

5.4.1 Selecting a Move

After the game engine has generated the next level of board positions and added them to the tree, the root is passed to the agent by calling the `selectMove` function. The core of this function is shown in Figure 5.5.

```
Node *current, *best;
double max  = -1;

best    =   NULL;
current =   root->child;

//      Step through each board position.
for (int q=0; q<num; q++){
      //      Convert the board so it can be fed into the network.
      double *convertedBoard = convertBoard(current->data, turn);
      network->feedForward(convertedBoard);   //    Feed board into network.
      double *output = network->getOutput();  //    Get the output array.
      current->data->value = output[0];       //    set the value of the board.

      //      Set this board as our selected move if it has the highest value.
      if (current->data->value > max){
            max = current->data->value;
            best = current;
      }

      //      Get the next board.
      current=current->next;
}
```

Figure 5.5: `moveSelection` code segment.

Basically, the agent converts each board into a format suitable for its particular network. Each of these converted boards is then passed to the network. The network's output is stored in the `value` variable of the board's structure. The actor is now able to use these values to select which move should be performed. Formally, the network represents the agent's current policy π, and the value it produces is the value V of the state s (the current board) at time t using that policy. Thus, the board is assigned the value $V(s_t)$.

In the discussion of reinforcement learning the need for the actor to use an evaluative feedback method such as softmax for action selection was discussed (3.2.3). Using one of these methods allowed an agent to explore actions that start with initially poor values. However, due to the agent in this implementation using a neural network for function generalisation the TD-error affects the entire network. This is because, most if not all of the network's nodes will have fired to some degree in performing the evaluation and therefore, they will all receive a small level of adjustment from any TD-error. Therefore, as all the weights are being adjusted when the greedy action is selected, there is no further need for exploration.

However, in section 5.3.2, a problem with repeated sequences of moves was discussed. To help reduce the occurrence of this problem the agent stores what the last move was and if the selected move is the same as the previous turn for that player then a random move is selected from the remaining legal moves. Therefore, the method actually implemented is similar to the _*-greedy* action selection method (3.2.3.2).

5.4.2 TD-error Calculation

After the agent has selected its move it updates the network by calculating the TD-error. This is performed in the `updateNetwork` function, which takes three parameters; the selected board position, the reward allocated by a terminal state and which players turn it currently is. The agent learns during the white players turn by using the reward value if there is one or the value associated with the root. The equation being used in this calculation is the TD-error function in the actor-critic method shown again here, where _ is the TD-error, *r* is the reward, _ is discount rate with the value 1, *V(s)* is the value of state *s* and *t* represents the current time step.

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Equation 5.1: TD error calculation in actor-critic method

Figure 5.6 illustrates this process. Basically, the board position selected by white is stored. Black then selects their move using the `selectMove` method described above. Now, after white has selected its next move, the newly

selected board positions' value is used as reinforcement to the network. Therefore, if after black has moved, it has become apparent that the best board position for white is a lot worse than the last board position then the network is adjusted to prevent the likelihood of the previous move being selected again. However, if it showed that the move led to a winning position then it would increase the likelihood of selection in the future.

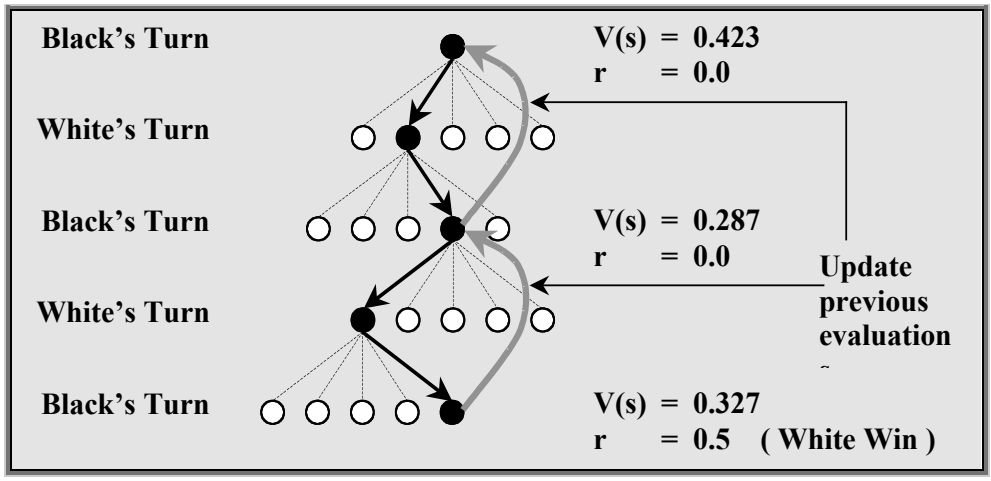


Figure 5.6:Illustration of network update.

Figure 5.7 shows the `updateNetwork` method. It can be seen here that Equation 5.1 above does not appear to be implemented as defined. Firstly, due to chess being essentially an episodic task the variable _ is always 1 (3.2.2), and therefore, has been omitted. Secondly, because we feed the last board position into the network, giving us our $V(s_t)$, prior to passing our TD-error we are effectively performing the value difference calculation $V(s_{t+1})$-$V(s_t)$ during the neural networks backpropagation. Finally, if it is not a terminal state the value of r is zero and therefore not used. Likewise, if it is a terminal state then the value of $V(s_{t+1})$ is also not relevant.

```
//      Only learn on whites turn.
if (turn == WHITE){

      //      Use the reward if there is one else use the current root nodes value.
      if ((r < -0.01) || (r > 0.01))
            TDerror[0] = r;
      else
            TDerror[0] = root->data->value;

      //      Call feedforward to set the output values for all nodes
      //      to the same values as when they generated their value
      network->feedForward(convertBoard(lastBoard, turn));
      network->update(TDerror);
}
```

Figure 5.7: `updateNetwork` code segment.

5.5 Neural Network Implementation

This thesis is primarily concerned with the application of reinforcement learning techniques to a large problem and not the various intricacies and variants possible within neural network theory. For this reason it was decided to implement the neural network using the simplest and most common approach. Thus, the network class uses a standard multilayer network using the *generalised delta rule* or *backpropagation rule* and an offset sigmoid threshold function (2.4).

The network class was written independently from the previous sections of code and tested against another network using the same techniques to ensure that it learnt in a similar manner. The class has been implemented at the layer level instead of the neural level to reduce the number of function calls and has been implemented to allow any form of standard network topologies. Each layer has a two dimensional array containing the sequence of weights and a threshold for each node. Each layer also has a link to the next layer and the previous layer allowing the resulting network to feed input forward and errors back appropriately. When first created the network initialises all weights and thresholds to a random number in the range from –0.1 to 0.1.

5.5.1 Processing Network Input

Once initialised the network can take an array of inputs, passed to the `feedForward` function. The number of elements in the input array should match the number of nodes in the input layer of the network. Each layer processes these inputs and produces an array of output values. If there is a subsequent layer then this output array is passed as input to the next layer.

The core of the `feedForward` function is shown in Figure5.8. It can be seen that the input layer simply distributes the input data by copying the input array into the output array. The hidden and output layers sum the weights times the input for all nodes. The threshold or bias is then added and the result is fed into the sigmoid function. Calling the `getOutput` function can also retrieve the output array.

```
//      The input layer has zero numWeights.
if (numWeights > 0){
      //      Sum the weights times the input, add bias and apply sigmoid function
      //      for all nodes in this layer.
      for (int i=0; i<numNodes; i++){
            output[i] = 0.0;
            for (int j=0; j<numWeights-1; j++){
                  output[i] += x[j]*weights[i][j];
            }
            //      Add bias.
            output[i] += weights[i][numWeights-1];
            //      Apply sigmoid threshold function to totals.
            output[i] = 1/(1+exp((-K)*output[i]))-0.5;
      }
}else{
      //      As this layer is an input layer simply copy values across to the output array.
      for (int i=0; i<numNodes; i++){
            output[i] = x[i];
      }
}
//      If this is not the output layer then pass this layers results on to the next layer.
if(nextLayer != NULL)
      nextLayer->feedForward(output);
```

Figure 5.8: `feedForward` code segment.

5.5.2 Backpropagation

The purpose of the neural network is for it to adapt its weights and the threshold of each node in order to learn. This is accomplished in the `update` function shown in Figure 5.9. When processing the output layer this function first must calculate the difference between the desired or target value and the actual outputted value for each node. This converts the values in the array *d* to be error values, the same as they would be when received by a hidden layer.

```
//      If this is the output layer then find the difference
//      between the target and the output.
if (nextLayer == NULL){
      for (int i=0; i<numNodes; i++)
            d[i] -= output[i];
}
//      For each node in the layer calculate the value of delta.
for (int i=0; i<numNodes; i++){
      delta[i] = K*(0.25 - output[i] * output[i])*d[i];
      //      Calculate the value of error value and adjust each weight for every
      //      every node in the layer.
      for (int j=0; j<numWeights-1; j++){
            err[j] += delta[i]*weights[i][j];
            weights[i][j] += gain * delta[i] * input[j];
      }
      //      Adjust threshold for each node.
      weights[i][numWeights-1] += gain * delta[i];
}
previousLayer->update(err);
```

Figure 5.9: update code segment.

The `update` function then calculates how much of the error should be applied to each node in the layer. This is used in turn to calculate the total amount of error to be propagated back to the previous layer and to adjust the weights of each node. Finally, the process is repeated for each layer in the network.

5.6 Individual Agent Models

There are numerous elements in this algorithm that affect the ability of a particular agent to learn. The optimal value and configuration for these is not known and therefore a number of agents have been developed to test different setups. Each of these agents has been implemented as a subclass of the TD-Agent class with different setup parameters. In addition the `convertBoard` function has also been redefined for each. This function takes a board position and converts it into an input array that suits the agent's neural network. The type of input array being used is determined by the interpretation of the state being used in that agent. These are described in more detail, along with details of variations in agent setups in the following subsections.

5.6.1 State Views

As described earlier, the purpose of the agent is to get relevant information about the current state of the environment and to use this information to determine an action. The difficulty here is finding a method for representing this state information that describes the relevant information in a form suitable for the agents neural network. Below is a list of some examples of what may be relevant state information.

- What pieces are on the board?
- The position of each piece on the board.
- The locations that each piece can move to in subsequent moves.
- The relationships between pieces.
- Whose turn it currently is.
- How many moves are there available to each piece.
- What colour square is a particular piece on?
- How many moves have occurred since the start of the game.
- What move was taken leading up to the current state.
- Is (kings or queens) castling still allowed?

The above list would require a very large input array to describe fully and would be too unwieldy and difficult to train as a single agent. Also, some may not be relevant or the network may be able to derive them itself from a well-constructed input array. Therefore, four separate input arrays have been used using some of the above examples of state information.

5.6.1.1 Piece-Weighted State View

The first type state representation was to simply identify which pieces are currently on the board. The network has an input node for every piece possible in the game. Therefore, for this project an agent using this view of the state would have eight input nodes. Each node would receive a one or zero informing the network that the piece is on or off the board, respectively.

This view is expected to generate an agent that can recognise board positions with piece advantages. It should also learn that various pieces are of more importance than others. However, it would not develop any ability to learn more subtle variations in play, such as positional advantages.

5.6.1.2 Positional State View

The positional agents also have a relatively simple view of the boards state. Essentially, we want to describe the specific location of each piece on the board. The network then should use the pattern classification and recognition techniques, which neural networks are particularly suited for, to learn what is a good board position.

These agents have a network that has eight input nodes, using the endgames created for this thesis, for each of the 64 squares on the chessboard. Thus, the network would consist of 512 input nodes in total. Each of these nodes receives a one or a zero informing the network that a particular piece is currently on that square or not. Therefore, at least seven of the nodes for each square will receive a zero. Squares that are empty will receive a zero for all eight inputs for that square.

It is hoped that agents using this state-view will be able to group boards with similar patterns together and give them appropriate values. The values that it assigns represent the pattern of the pieces and the importance of their locations on the board. These agents may also be able to identify similar information that the piece-weighted agents receive as they are effectively also receiving this information just distributed over a number of inputs.

5.6.1.3 Mobility State View

The problem with above types of agents is that they receive no information about where a piece is able to move to which may be of importance. The third type of state representation adapts the above method by including a second input node for each piece, at each square. Therefore, there are 16 input nodes per square and 1024 overall. The first node still identifies whether the piece is on that square. The second node receives an input of one if the piece can move to that square on the next move otherwise, it receives a value of zero. This method could easily be extended to indicate if a piece could get to the square in two moves or more by giving the input node values between zero and one, where a smaller value indicates more moves required before the piece could reach that square.

This input type has the advantage of clearly identifying for the network information about the movement and mobility of each piece and thus the rules of chess. It also conveys some information on the possibility of capturing pieces or the risk faced by one of its pieces from the opposition. It is believed that this type of input could allow the agent to learn more detail about various subtle elements of the game. It also provides the potential for the agent to be able to interpret the viability of a position by recognising patterns of future positions. The problem with this format is that it requires doing a 2-ply look-ahead search for every possible board position, which significantly increases the computational requirements. In addition to this, the agent has a larger network topology that requires significantly longer training games.

5.6.1.4 Relational State View

Chess is not just a game about the location of pieces on the board. A human player can often spend a lot of time working out which pieces are attacking and defending other pieces. For instance, they may try to ensure that each of their own pieces is defended by more pieces than opposition ones attacking it. Alternatively, if the defender/attacker ratio is the same then the defender may try to ensure the value of the piece defending are less than the value of the attacking pieces ensuring any exchange will be to his or her advantage.

The relational state view addresses this by having an input node for each relationship in the game. There is also an additional two input nodes used to indicate whether it is whites or blacks turn. This is required because there is no difference in the boards' representation when it changes turn. Therefore, there are $n(n-1)+2$ input nodes, where n is the number possible pieces in the game.

5.6.2 Agent Variations

Using different forms of input is only one facet that could provide different levels of learning. Another is by altering various aspects of the processing within the agents. The variations investigated in this thesis revolve around altering the network topology and learning rate, adding noise to the networks inputs and limiting the length of training games. These are discussed in further detail in this section.

5.6.2.1 Standard Agents

The first group of five agents have a learning rate of 0.01; this is a common starting rate as it provides a reasonable learning speed. They each also have a maximum game length of 5000 and have no noise added. The first three agents, using the first three state-views described above, are using a single hidden layer containing 300 hidden nodes as their network topology. The fourth agent, using the relational state-view, and the fifth, which is also using the positional state-view, have two hidden layers with 300 and 20 nodes in each.

5.6.2.2 High Gain Agents

The above learning rate is fine for most neural networks; however, an agent may only see particular board patterns infrequently. Therefore, the small learning rate may not be enough to allow the agent to learn. Thus, the above five agents were re-implemented using a learning rate of 0.1. This should speed learning initially but may also cause the agent to repeatedly jump from one side to the other of the optimal solution, thereby, never reaching the best result.

5.6.2.3 Noisy Agents

One problem with a deterministic game such as chess is that there is no uncertainty. Yet this is an intrinsic element of the control problem from which TD emerged. For instance, sensors and actuators generally have a level of inaccuracy in their measurements. Also, games where TD has been successfully applied such as Backgammon generally contain a non-deterministic factor; for instance moves in Backgammon are defined by a throw of the dice.

For these reasons it was thought that adding noise to the agents' perception of the state could simulate the above uncertainty and therefore, may improve learning. Thus, the standard agents were re-implemented again with a random fluctuation of 0.2 for each input. For example, if the input should be 0 then it would receive a number between 0 and 0.2, and likewise, an input of 1 would receive a number in the range 0.8 to 1.

It was also required in these agents to alter their `updateNetwork` function to ensure that the random noise simulated when they first selected the move is the same when the agent re-applies the inputs to set the networks' internal weights ready to receive the TD-error value. This was done by storing the inputs generated initially, with the last board position, and using these rather than producing new ones.

5.6.2.4 Limited Length Games

It was mentioned earlier (5.3.2) that games were set to terminate after 5000 moves. However, the length of games affects the frequency that a terminal state is reached, which in turn may affect the ability of the agent to learn. The shorter a game the less the number of intermediate states that rewards must propagate back through. In order to test this, another five agents were implemented based on the standard agents but had their games limited to a maximum of 100 moves per game.

5.6.3 Agent Details in Brief

The twenty individual agents have been listed in Table 5.2 in order to clearly identifying their details. They have been broken into 4 groups of five, where the 4 groups relate to the agent variations. The abbreviated names, listed in the table, will be used during the remainder of this thesis when referring to individual agents.

Variation	State View	Attributes					Abbreviated name
		# of Input Nodes	Learning Rate (_)	Length of Games	# of Hidden Layers	Noise added	
Standard	Positional_1	512	0.01	5000	1	no	Std_P1
	Mobility	1024	0.01	5000	1	no	Std_M
	Positional_2	512	0.01	5000	2	no	Std_P2
	Relational	58	0.01	5000	2	no	Std_R
	Piece Weighted	8	0.01	5000	1	no	Std_PW
High Gain	Positional_1	512	0.1	5000	1	no	HG_P1
	Mobility	1024	0.1	5000	1	no	HG_M
	Positional_2	512	0.1	5000	2	no	HG_P2
	Relational	58	0.1	5000	2	no	HG_R
	Piece Weighted	8	0.1	5000	1	no	HG_PW
Noisy	Positional_1	512	0.01	5000	1	yes	N_P1
	Mobility	1024	0.01	5000	1	yes	N_M
	Positional_2	512	0.01	5000	2	yes	N_P2
	Relational	58	0.01	5000	2	yes	N_R
	Piece Weighted	8	0.01	5000	1	yes	N_PW
Shortened	Positional_1	512	0.01	100	1	no	Short_P1
	Mobility	1024	0.01	100	1	no	Short_M
	Positional_2	512	0.01	100	2	no	Short_P2
	Relational	58	0.01	100	2	no	Short_R
	Piece Weighted	8	0.01	100	1	no	Short_PW

Table 5.2: Individual agents and their details.

5.7 Training

The agents were trained by giving them a large number of randomly selected starting board positions (these positions were designed to all be legal board positions), which they played until a terminal state or the maximum number of turns had been reached. In each case both the white and black player used the same agent for deciding on move selection but the agent only learnt on the white player's turn. This section explains the reasoning behind this choice of training.

The type of endgame selected was for each player to have a king, rook and pawn at the start. Therefore, as there is a pawn, each agent must also allow for the presence of a queen (1.2.2). Each initial board position is selected randomly as it partially resolves two problems. Firstly, during development it was found giving preset starting positions could both limit the possible or likely states, thereby, limiting the agents experience to those states actually visited. Secondly, with any given starting position some agents, due to their randomly selected starting weights, are unable to complete the games within the maximum number of turns allowed.

During each game it was decided that the agent should play both sides. This prevents the inclusion of outside knowledge into the system that would occur if another chess program or human played the black player. A random player for black was not used for two reasons. Firstly, because a random player was used for testing and secondly, because as the agent gained experience then its competitor, itself, would as well, allowing even further improvement.

Finally, the agent has been restricted to only learn during the white players' turn in order to ensure the agents' network has the same weights and threshold values as when it first evaluated the given board, thus producing the same board evaluation. If it also learnt during blacks turn then the networks weight would have been altered during blacks turn and would now produce a different value to that generated when it first selected the board.

5.8 Conclusion

This chapter has described in detail what design decisions and algorithms have been used in order to achieve the aims identified in the introduction. The implementation consists of three primary components: the game engine, TD-agent and its associated neural network. Finally, a number of agents were implemented, each with various views of the state information and with different parameters. The creation of a number of different agents allows this thesis to investigate which combination of factors best allows learning when the TD algorithm is applied to the problem of chess.

CHAPTER 6 Results and Discussion

Modern chess is too much concerned with things like pawn structure.
Forget it, checkmate ends the game.

Nigel Short

The original purpose of this project was to investigate reinforcement-learning techniques when applied to chess endgames. The previous chapter described the implementation of the TD-agents employed in this thesis. Each of these agents had different views of the state space or variations to their parameters that affect the agents' ability to learn. It also described the training methods used by these agents.

At various stages throughout the agents training and after training was completed they were tested to gauge if they had learnt anything about how to play chess. As previously stated the agents were not expected to learn to play competitive chess for two primary reasons. Firstly, they were not using any form of lookahead search and, secondly, because of the extremely information-sparse environment. Therefore, they were only tested for some evidence of learning.

This chapter will first outline the data collected by the agents. Two main elements were then investigated and discussed in order to identify the various agents' progress. The first aspect investigated, concerned their ability to differentiate between a good and bad board position. This was achieved through testing a number of known terminating states and comparing them. Secondly, the agents' ability to play against a random player was tested. Prior to training, the agents did know what moves to make and thereby, simple moved pieces randomly. Therefore, as they were trained they learn more about the moves being made and improved their choice. Consequently, it was predicted they would improve their performance, giving a better win loss record against the random player.

6.1 Data Collected

It was decided that the agents should be tested at regular intervals during the training process in order to build up a progressive picture of their development. In addition to the testing data, information was also stored concerning the games played during the training process, providing information about the agents' learning process. Finally, backup copies of the agents were saved after every 500 training games allowing the agent to be loaded and tested again at a later stage. This section will describe what data and testing was performed during training.

Firstly, general information about training games was recorded after every 10 training games. The results of these games are interesting because the agent was playing against itself. The information collected from these games consisted of the number of white wins, black wins, number of cut-off games and the number of draws. In addition, a record was kept of the number of moves taken in each game and the average number of moves over the last 10 games.

The first part of the agents' testing phase was performed after every 10 training games. During this phase a set of 12 handpicked board positions (Appendix C lists these positions) was fed through the agent twice for evaluation, which was then recorded. The first evaluation was with the agent viewing the board as the white player, the second as the black player. All of these board positions represent various terminating states. The set contains 4 white winning positions, 4 draw/stalemate positions and 4 black winning positions. This provides data about how the agents view various states and provides the first information about when and how effectively the agents are learning. The terminating positions were predicted to be the first set of states the agents' should learn, as these are the first states to receive true and accurate reward information in the TD algorithm.

The second testing phase involved getting the agents to play a set of games against a random player. It was decided to play two sets of 50 games after every 500 training games. The first set was played with random starting positions (described in appendix D), the same as with the training games. The second set was played against a preset starting position (described in appendix E). Information concerning the number of white and black wins, draws, cut-offs and number of moves was recorded about each set.

6.2 Number of Games Completed

During the training process each agent was given similar amounts of processor time in order to insure they each were compared on a reasonably even playing field. However, each agent performed at significantly different speeds resulting in large variations in the number of training games. The difference was caused primarily by three factors. The first was that each of the different state-views required big variations in the amount of processing required (5.6). Secondly, agents with larger neural network topologies tended to run slower as there were more nodes to propagate results through. Finally, due to the random weights on the networks nodes at start up, some agents could get stuck in long sequences of repeating moves making them play much slower.

This variation in the number of training games played means that they have been compared against each other in the rest of this chapter on an unequal footing. Therefore, some agents would have compared better had they played an equal number of games. However, the slow agents could not be given more processing time due to hardware and time limitations. Table 6.1 gives a listing of the number games played, number of games cut-off before completion and the actual number of games finished that provided training for each agent.

Agent	# of Games	# of cut-offs	# of finished games	Agent	# of Games	# of cut-offs	# of finished games
Std_P1	4190	911	3279	N_P1	9180	1	9179
Std_M	3860	1273	2587	N_M	5630	0	5630
Std_P2	4320	955	3365	N_P2	21140	0	21140
Std_R	10500	6497	4003	N_R	33750	5430	28320
Std_PW	100000	13477	86523	N_PW	55800	13503	42297
HG_P1	5380	976	4404	Short_P1	45500	40885	4615
HG_M	2260	683	1577	Short_M	38720	31964	6956
HG_P2	3470	908	2562	Short_P2	48710	44693	4017
HG_R	23320	6650	16670	Short_R	100000	93245	6755
HG_PW	96730	18666	78064	Short_PW	100000	71357	28643

Table 6.1: Number of training games played by agents.

6.3 Terminating Positions

Over time it is expected that each agent would learn to assign higher values to board positions representing white wins, as opposed to those that identify draws or black wins. In this situation it is not relevant what the actual value is assigned, just that it is higher. Figure 6.1 shows how the agent Short_M learns three-example board positions overtime: a white winning position (white line), a draw position (grey line) and a black winning position (black line). It can be seen that this agent has successfully ordered the random starting values into the correct ordering.

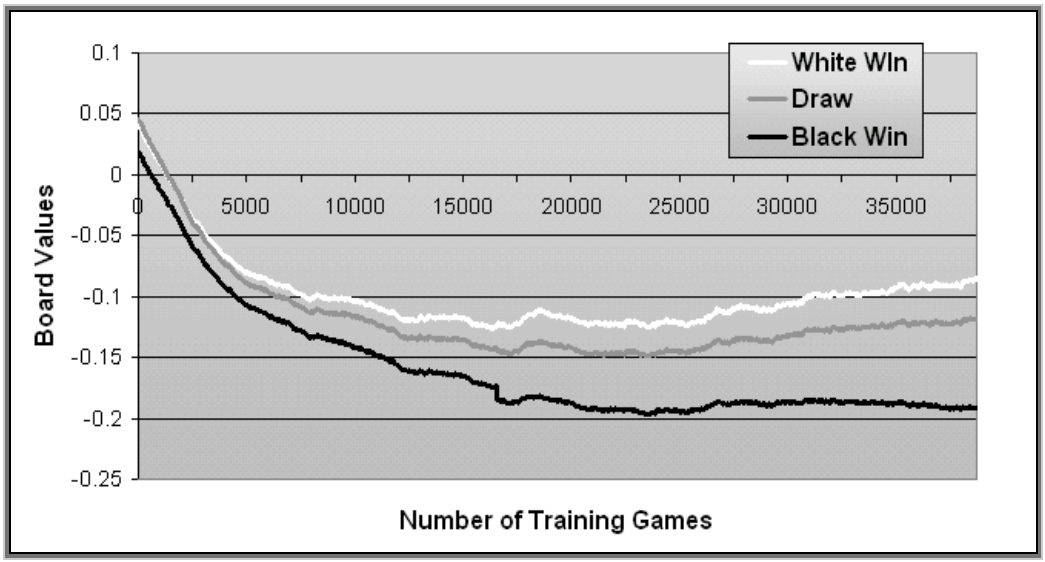


Figure 6.1: Short_M learning terminal state board positions overtime.

However, this graph only identifies that it learnt these particular positions, not all terminal states in general. Figure 6.2 shows the same agent again but in this graph the minimum board value through to the maximum is shown for each type of terminal state. It can be seen that the terminal states for a white victory are clearly identified as having a higher value than other terminal states. However, while it has placed draws higher than black wins on average there is no clear gap between them. This lack of distinction is a result of the combination of two factors. Firstly, there was only a small difference between the rewards awarded to these two states. Secondly, because the black player won training games very infrequently, the agent did not see these board positions enough to fully learn a value for them.

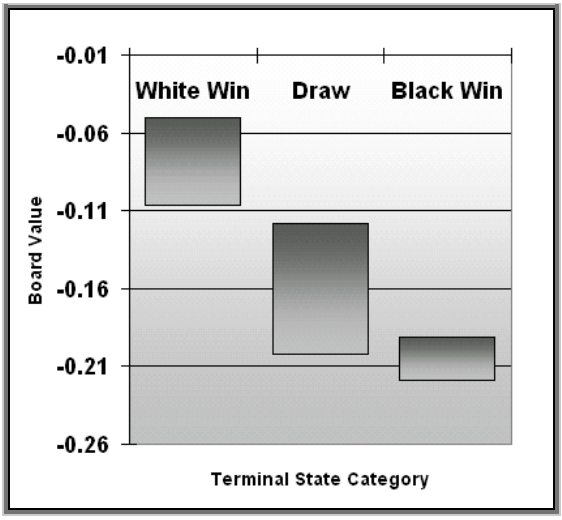


Figure 6.2: Short_M Comparison of all terminating Positions

In the above graphs the agent Short_M was used primarily because it highlighted the case of an agent that had successfully learnt some aspects of the terminal states. An example of an agent that learnt to distinguish between all three types of terminal states is Short_PW, shown in Figure 6.3. Not only does the agent rate white wins higher than draws and draws higher than black wins, it also has a clear gap between them all.

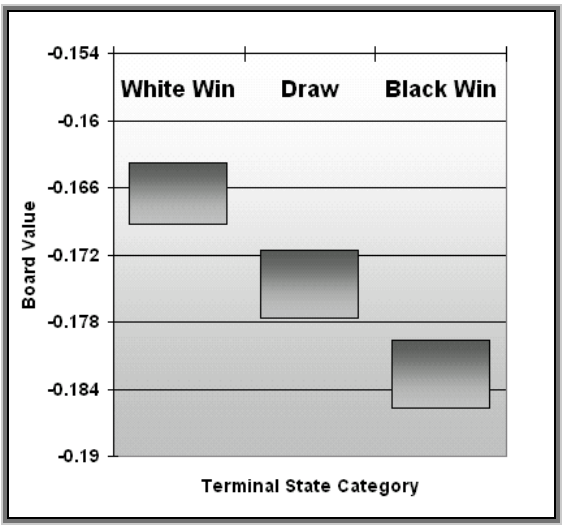


Figure 6.3: Short_PW Comparison of all terminating positions.

However, not all the agents were able to duplicate these results to the same degree. Appendix D provides similar graphs showing the terminal state evaluations for each agent. It can be seen in these graphs that a number of agents failed to learn any useful information about terminal states. Table 6.2 shows a summary of which agents learnt various aspects of the terminal states. The list below the table describes the meaning of each row.

	Std_P1	Std_M	Std_P2	Std_R	Std_PW	HG_P1	HG_M	HG_P2	HG_R	HG_PW	N_P1	N_M	N_P2	N_R	N_PW	Short_P1	Short_M	Short_P2	Short_R	Short_PW
A	×	✓	×	×	✓	✓	✓	×	×	✓	✓	×	×	✓	✓	×	✓	×	×	✓
B	×	✓	×	×	✓	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	✓
C	×	✓	×	×	✓	×	✓	×	✓	×	×	×	✓	×	×	✓	✓	×	✓	✓
D	×	×	×	×	✓	×	✓	×	×	×	×	×	×	×	×	×	×	×	×	✓

Table 6.2: Analysis of terminating board positions for all agents.

A. This row indicates agents that generally, but not always, placed white winning positions higher than the draws and black wins.

B. The second row indicates those agents that placed white winning positions with a clear gap separating them above both draws and black wins.

C. This row shows which agents generally, but not always, placed draws higher than black wins.

D. The final row shows agents that placed draw positions above black wins with a clear gap separating them.

The above table plainly identifies the agents and their level of success in learning terminal states. Clearly, the short_PW and Std_PW agents showed the greatest ability to learn terminating positions. This is mainly because any variation in the inputs had significant influence on whether it measured a board as being good or bad and they were not required to learn complex patterns. This lack of a complex pattern to learn could have caused the HG_PW agent to bounce wildly, which may be why it failed to learn. The other group of agents that performed well were the mobility agents: Std_M, HG_M and Short_M. These were also able to learn the majority of the required separations. This is most likely because they received a lot of information about the state. Their performance, however, was a surprise due to them being the slowest at playing and therefore having the least amount of training games.

It is also clear from the table that the positional and relational state-views showed no ability to differentiate between board positions. The single tick that some of them received is most likely through natural random fluctuations than through any true learning ability on the part of the agent. Interestingly, it can also be observed that when using a noisy input array even the piece-weighted and mobility agents were unable to learn, which goes against existing theory, that agents perform better in deterministic environments when noise is added.

6.4 Performance against a random player

If an agent can differentiate between terminal state positions then eventually these board ratings should start to propagate back up the game tree improving earlier states. Once these states have improved enough then, theoretically, the agent should begin to improve its play against a random player. Therefore, the agents that are most likely to have learnt to defeat a random player are those that performed well with the terminating state analysis, such as the piece-weighted and mobility agents.

Figure 6.4 shows Short_M's performance against a random player. The Polynomial trend line clearly shows that the agent has improved its play during training. However, both of the piece-weighted agents, Figure 6.5 and 6.6, show no discernable improvement or their performance worsened over time. Therefore, they were able to learn the difference between various board positions but this did not help their actual play. This was not unexpected because when they were learning to interpret between good and bad states, they were only looked at which pieces were on the board. Therefore, they could not interpret any states in between and thus did not learn to play well.

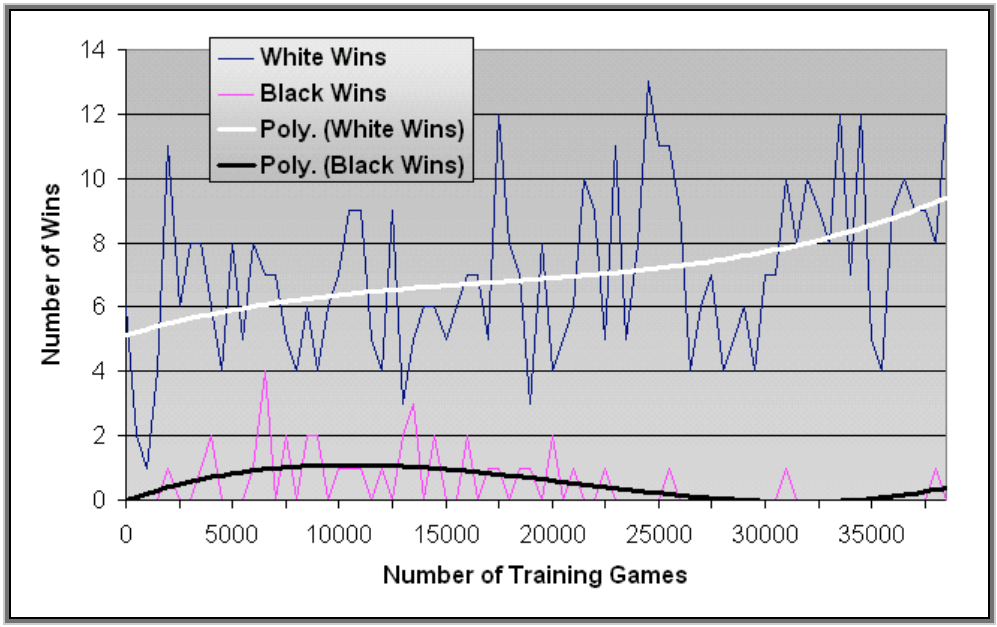


Figure 6.4: Short_M playing random player using the preset starting position.

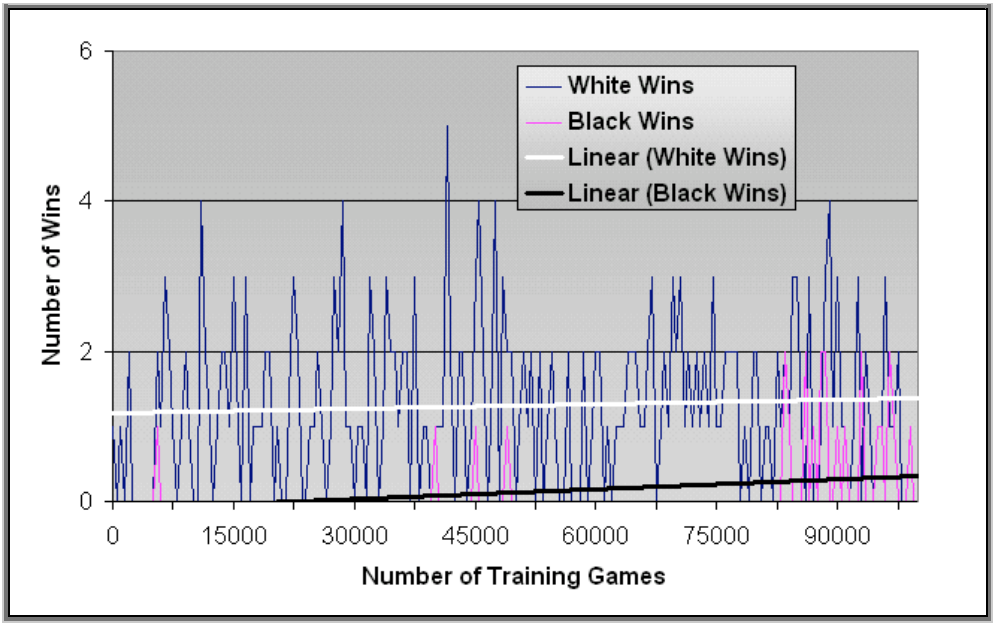


Figure 6.5: Short_PW playing random player using the preset starting position.

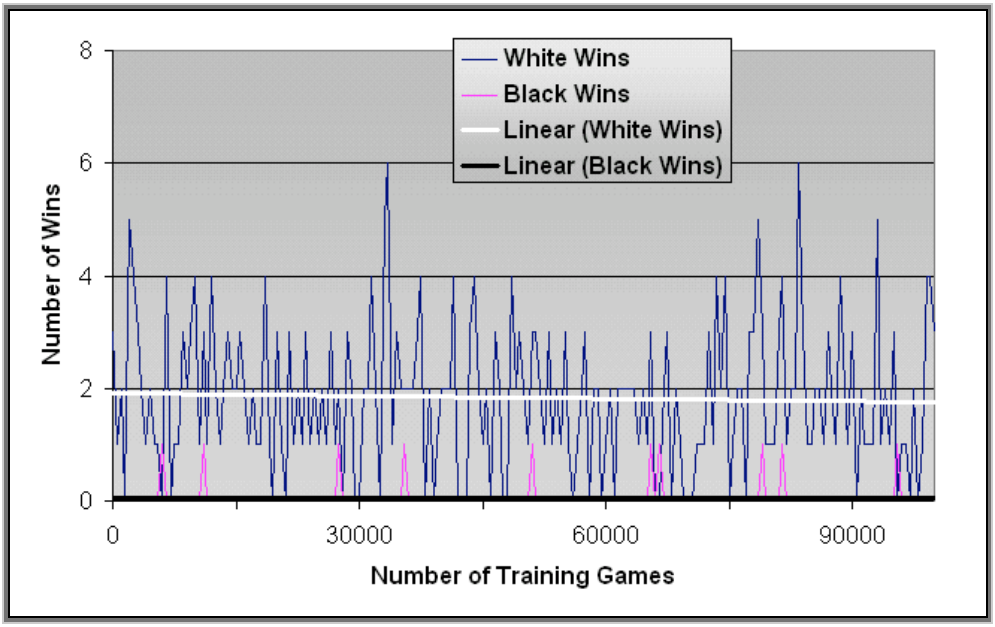


Figure 6.6: Std_PW playing a random player using the preset starting position.

It may also be noticed in all of the above graphs, that the white player tended to win more from the outset. This is because these graphs are using the preset starting position, which may have a small advantage for the white player. The advantage to white in this situation does not really matter because we are only interested in measuring the improvement over time. However, the advantage to white may have allowed the agent to appear like it had learnt more than it had.

The games using random starting positions would be expected to not have the initial advantage for white in the above graphs. Figure 6.9 shows another graph of Short_M, using random starting position, where once again it can be seen that it has generally improved its play overtime and this time both white and black started on mutually even grounds. The white player starts with more wins, more through general luck this time, rather than through some inherit advantage built into the starting position. However, its performance fluctuates as some games initially favour one player over the other. These large fluctuations exist in most of the other agents when using random starting positions resulting in little information being able to be derived from them.

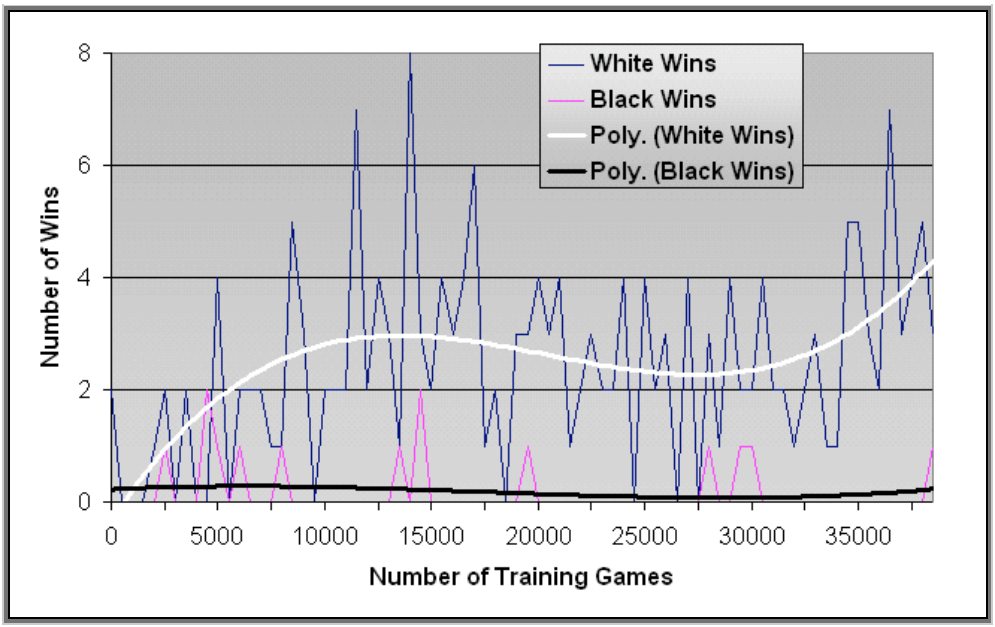


Figure 6.7: Short_PW playing a random player using random starting position.

There were two additional agents that stand out as having showed at least some small indications of improving overtime. The first, Std_M shown in Figure 6.8, showed that it may have been improving but, due to it playing very few training games, there is only a very small amount of data available, making any improvement difficult to claim definitively. The second was Short_P1 illustrated in Figure 6.9, which had showed no real evidence of learning the terminating positions (6.3). However, its improvement against a random player may be caused by an anomaly of a couple of extra wins in the last few rounds rather than a general trend upwards.

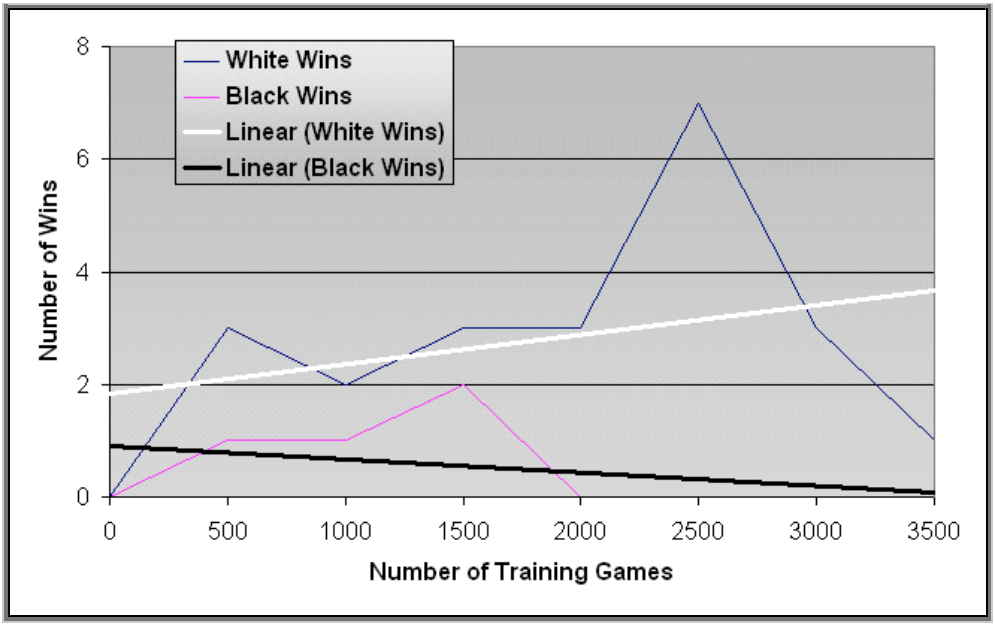


Figure 6.8: Std_M playing a random player using random starting position.

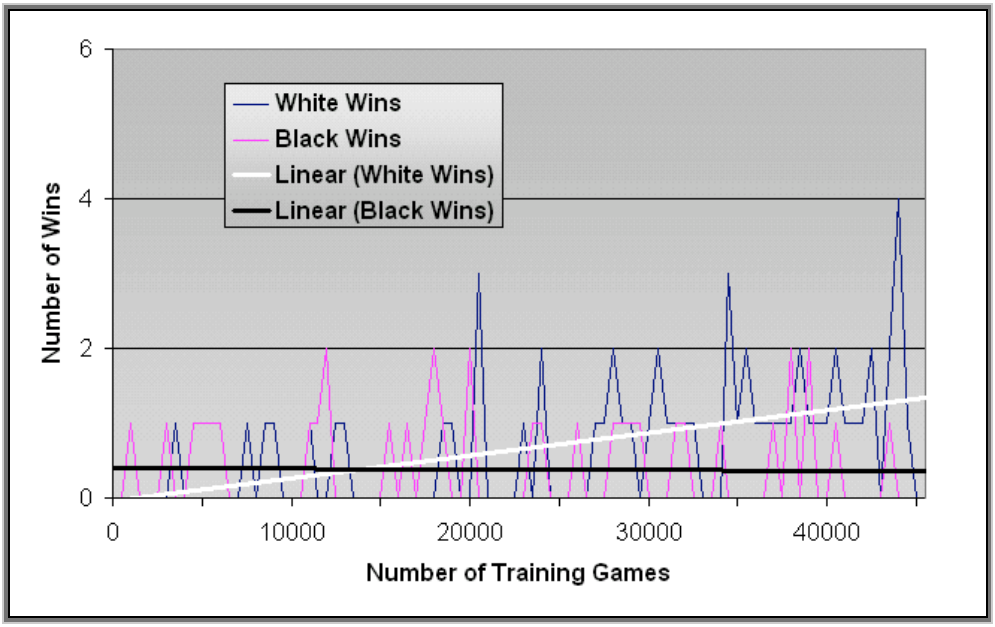


Figure 6.9: Short_P1 playing a random player using random starting position.

Appendix F gives all the other agents graphs for both random and preset starting positions. It can be seen that the majority of the agents failed to show any real improvement on both random and preset starting positions.

CHAPTER 7 Conclusion

The most important feature of the Chess position is the activity of the pieces.

Michael Stean

Reinforcement learning is a relatively new field of research that has gained popularity over the last ten years as a means of creating agents capable of learning behaviour with little or no external knowledge through interaction with an external environment. Previous work in this field has shown great promise when applied to non-deterministic environments, problems with small state spaces or when some external knowledge has been used to assist training. However, there has only been limited work done on testing reinforcement learning applied to deterministic problems with large state-spaces while using no external knowledge.

This thesis has investigated the performance of reinforcement learning when applied to an information-sparse environment by implementing a number of agents and training them to play chess endgames through self-play. Chess presents a deterministic problem that has a huge state space and has previously had chiefly brute force approaches applied. The aim was not to play competitive chess but to demonstrate that these agents were able to show some learning ability in such a sparse environment.

Each of the agents implemented in this project used a different view of the state information or variations in parameter settings in order to find an agent that learnt reasonably well. These agents were then tested in two ways. The first was to gauge whether they were able to determine the difference between good and bad terminating positions. This was expected to be where the agents would first show any learning ability, if they were going to show any at all, as these states receive rewards directly from the environment. The second was to measure their performance in playing a series of games against a random player. For them to achieve success at this level the agents would have needed to propagate the terminating state results back through earlier positions.

7.1 Discussion

It was shown in chapter 6 that the Std_M, Std_PW, HG_M, Short_M, and Short_PW agents showed clear and definitive evidence of learning to differentiate between terminal state positions correctly. It can also be noted that because the successful agents were all using the same two state views that their success was not just a statistic anomaly. Therefore, clearly the TD(_) algorithm was capable of learning in this sparse environment.

The piece-weighted agents performed the best but they also had the easiest task as they could easily detect changes in the number of pieces on the board. The mobility agents' high performances, while not as strong as the piece-weighted agents, were particularly impressive as they had significantly less training games. Their main success came from the extensive state information provided, as they knew about how the pieces moved and interacted. This as the quote at the beginning of this chapter stated, is the most important feature in the game of chess.

The second test showed at what level of knowledge the successful agents were able to learn. The algorithm is designed to propagate results back up the game tree, providing the opportunity for the agent to learn a weighting for all states. However, this process can take a long sequence of training games to accomplish and therefore, the agents were not expected to show much improvement at this level.

Section 6.4 showed that the Short_M agent did show quite a strong improvement using both random and the preset starting position. In addition, it accomplished this with less than 7000 completed training games. This result shows that this algorithm does hold the potential to learn behaviour in very complex environments. Nevertheless, the fact that none of the piece-weighted agents improved highlights how important a good representation of the state information is to the algorithm's ability to learn behaviour.

7.2 Problems

It can be seen, from section 6.4, that the HG_M and Std_M agents, which like the Short_M agent also used the mobility view, did not show the same level of improvement. The difference between these agents, which highlights the primary problem encountered, is the length of the training games before being cut-off. This problem caused two issues during training. The first was that the longer training games resulted in the agents played significantly less games, giving them fewer examples of various states. The second problem is that an excessively long sequence of moves in a game that is not cut-off, such as 4000 moves, can hamper the agent's ability to learn significantly during the early stages of training. This is because after each move the neural network is updated a small amount, which might be in the wrong direction, and after a long sequence of moves these small changes can accumulate to a point that exceeds the effect of the true reward received from the terminating state. Therefore, the agent can potentially head off on a tangent preventing it from learning.

The solution to this problem that was used in the shortened versions of the agents was to cut off games after only 100 moves. This, as the Short_M agent shows, did improve the agent's ability to propagate the terminating state results up the game tree. However, it presented the second problem of there not being enough games that reached a terminating state. Therefore, even though Short_M played 38720 games, less than 7000 actually contributed to the agent's training.

The final problem was that when a training game did complete it was usually a draw. Therefore, the agents received relatively few examples of white and especially black wins. This was potentially why most of the agents were unable to learn terminating positions. The best way to get around this is to simply have many more training games, which was not possible in the time available.

7.3 Future Work

This thesis has shown that clearly the agent will learn in this environment, however, to get more impressive results the issue of deep game tress would need to be addressed more directly than in this project. While there are a number of variations that could be implemented in future work, this section will briefly outline three suggestions.

The most straight-forward suggestion would be to simply use a higher _ value. This has recently, after this project was virtually completed, been shown to improve learning when used in correlation with a neural network for function generalisation. It would allow the rare occurrence of terminating state position results to feed back further affecting more of the network. However, this could also increase the effect of the erroneous updates during long sequences of moves and so should be used in correlation with other methods of reducing the length of games.

Future work would almost certainly need to ensure that the sequence of moves is minimised, most likely through artificial means. One possible way could be to present starting positions that are actually also terminating positions. The game will then end immediately giving a reward. Agents could then learn the terminating position quickly without interference from earlier unlearnt states. After a period of time they could then be presented with starting positions that would require a couple of moves to reach a terminating state. This altering of the starting position could continue until the general random starting position is used. Once the random starts are being used in this situation the agent would have already learnt a great deal about the problem domain and would not tend to wander to the same degree as they did in this project.

A third approach for reducing the affect of these long sequences of random updates could be to introduce a new parameter into the equation. First, an average length of the training games being played would be maintained. Then, the TD-error could be multiplied by the inverse of this calculated average, times the number of moves already played in the sequence. When a terminating position is eventually reached the parameter is set to one giving the TD-error its

usual full affect. If the sequence is longer the parameter is capped at the value one. This process would significantly reduce the importance of the early moves causing only small updates but would still maintain a high update for terminating positions. This process could be extended further by slowly decreasing this parameter as it learns, so that the earlier positions gain more importance as training continues.

7.4 Conclusion

This thesis has set out to investigate whether the TD(_) algorithm, one method of reinforcement learning, using standard back-propagation neural networks for function generalisation, was able to learn any useful information when interacting with a large deterministic problem such as chess, without including any external knowledge. While, the project found that there are a number of issues that need to be carefully considered during implementation, it showed that the algorithm was capable of learning terminating positions. This thesis also illustrated that the TD(_) algorithm was able to propagate the terminating states results up the game tree, thereby, improving game play. Finally, it highlighted that the depth of the game tree must be prevented from becoming to deep, as this can hamper the methods effectiveness at learning, and provided some ideas on improvements to the methodology to avoid this problem in future work.

References

- Anon, “Chess-Poster”, site viewed 3/8/2001, URL- http://www.chess-poster.com/index.htm.
- Baxter, Jonathan., Tridgell, Andrew., and Weaver, Lex., “Learning TO Play Chess Using Temporal Differences”, *Machine Learning*, vol. 40, No. 3, 2000, pp. 243 - 263
- Beale, R., and Jackson, T., *Neural Computing: An Introduction*, Institute of Physics Publishing, Bristol and Philadelphia, 1990.
- Campbell, Murray., “Knowledge Discovery in Deep Blue”, *Communications of the ACM*, vol 42, No. 11, November 1999, pp 65 – 67.
- Fausett, Laurene., *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*, Prentice Hall, New Jersey, 1994.
- FIDE, World Chess Federation Network, created 1999. site viewed 8/11/2001. URL- http://www.fide.com/cgi-bin/material/main.pl.
- Hayes, Jean E., and Levy, David N. L., *The World Computer Chess Championship: Stockholm 1974*, Edinburgh, University Press, 1976.
- Kaelbling, Leslie P., Littman, Michael L. and Moore, Andrew W., “Reinforcement Learning: A Survey”, *Journal of Artificial Intelligence Research*, Vol. 4, 1996, pp. 237-285.
- Levy, David., Chess and Computers, Computer Science Press Inc, Woodland Hills, California, USA, 1976.
- Littman, Michael L. and Szepesvari, “A Generalized Reinforcement-Learning Model: Convergence and Applications”, *Proceedings of the 13th International Conference on Machine Learning*. Morgan Kaufmann, Bari, Italy.1996, pp. 310-318.
- Russell, Staurt., and Norvig, Peter., *Artificial Intelligence: A Modern Approach*, Prentice Hall, New Jersey, 1995.
- Samuel, A. L., “Some Studies in Machine Learning Using the Game of Checkers”, *IBM Journal of Research and Development*, vol. 44, No. 1/2, 2000, pp. 206.
- Schaeffer, Jonathan, “The Games Computers (and People) Play”, *Advances in Computers*, volume 50, Academic Press, editor Zelkowitz, M. V., 2000, pp. 189 – 266.
- Scott, Jay., “Quiescence search”, site viewed 29/5/2001, URL – http://forum.swarthmore.edu/~jay/learn-game/methods/quiesce.html.

- Singh, Satinder P., and Sutton, Richard S., “Reinforcement Learning with Replacing Eligibility Traces”, *Machine Learning*, vol. 22, 1996, pp. 123-158.

- Smith, Martin C., *Temporal Difference Learning in Complex Domains*. PhD Thesis, University of London, 1999.

- Sutton, Richard S. and Barto, Andrew G., *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: A Bradford Book, The MIT Press, 1998.

- Tesauro, Gerald., “Temporal Difference Learning and TD-Gammon”, *Communications of the ACM*, Vol. 38, No. 3, March 1995.

- Thrun, Sebastian., “Learning to Play the Game of Chess”, *Advances in Neural Information Processing Systems 7*, The MIT Press, Cambridge, MA, 1995, pp. 1069 – 1076.

- Zurada, Jacek M., *Introduction to Artificial Neural Systems*, West Publishing Company, StPaul, 1992.

Bibliography

- Boyan, Justin A., *Modular Neural Networks for Learning Context-Dependant Game Strategies*, Masters Thesis, University of Cambridge, 1992.

- DeCoste, Dennis., "The Future of Chess-Playing Technologies and the Significance of Kasparov Versus Deep Blue", *Proceedings of the AAAI-97 Workshop on Deep Blue vs Kasparov: The Significance for Artificial Intelligence*, July 1997.

- Furnkranz, Johannes., "Bibliography on Machine Learning in Strategic Game Playing", Austrian Research Institute for AI. Site viewed 29/5/2001, URL - http://www.ai.univie.ac.at/~juffi/.

- Lee, Kai-Fu., and Mahajan, Sanjoy., "The development of a World Class Othello Program", Artificial Intelligence, vol. 43, 1990, pp. 21 – 36.

- Levinson, Robert., "General Game-Playing and Reinforcement Learning", *Computational Intelligence*, vol. 12, No. 1, 1995, pp. 176 – 196.

- Mitchell, Tom M., and Thrun, Sebastian B., "Explanation Based Learning: A Comparison of Symbolic and Neural Network Approaches", Machine Learning: Proceedings of the Tenth International Conference, 1993.

- Sutton, Richard S., "Learning to Predict by the Methods of Temporal Differences", Machine Learning, vol. 3, 1988, pp. 9 – 44.

- Tesauro, Gerald., "TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play", Neural Computation, vol. 6, 1994, pp. 215 – 219.

- Walker, Steven., Lister, Raymond., and Downs, Tom., "A Noisy Temporal Difference Approach to Learning 'Othello'", *Proceedings of the Fifth Australian Conference on Neural Networks*, 1994, pp. 113 – 116.

APPENDIX A
Algebraic Notation

Algebraic notation is the standard system of notation recommended by FIDE for all tournaments and publications. First, each piece is given a capital letter to identify it: K = king, Q = queen, R = rook, B = bishop, N = knight. (Different letters can be used in languages other than English). Pawns do not have a letter and can be recognised by the absence of a letter. Secondly, the individual squares on the board are described by giving the eight files (from left to right for White and from right to left for Black) are indicated by the small letters, a, b, c, d, e, f, g and h, respectively and the eight ranks (from bottom to top for White and from top to bottom for Black) are numbered 1, 2, 3, 4, 5, 6, 7 and 8, respectively. Figure A.1 shows this coordinate system.

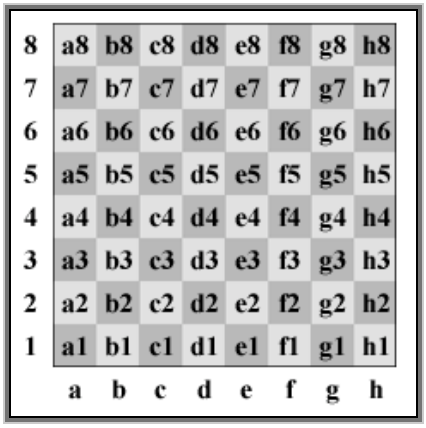


Figure A.1: Algebraic notation coordinate system.

Each move of a piece is recorded by following a number of rules, not all of which are described here, for a full description visit the FIDE site. Firstly, the piece being moved is identified by its corresponding letter, no letter is used when a pawn is moved. This is then followed by the coordinate of the square the piece is being moved too. If this move results in the capture of an opponent's piece then the letter 'x' is inserted between the pieces identifying letter and the coordinate. If there are two of the same type of piece that could have made the move and if prior to making the move they were in the same file then their rank prior to moving is identified after the pieces letter and before the 'x' or coordinate. Otherwise, their file, prior to moving, is recorded to identify which piece made the move.

APPENDIX B
Random Starting Positions

During training and some of the testing games a random starting position was selected. This appendix will list the rules applied to placing the pieces for these starting position.

- The white pawn was placed randomly in any square in the ranks 2, 3, or 4 (Appendix A).
- The black pawn was placed randomly in any square in the ranks 5, 6, or 7.
- Both the white and black rooks where placed randomly anywhere on the board with the condition they were not placed on a square containing any other piece.
- Both the white and black kings where placed randomly anywhere on the board with the condition they were not placed on a square containing any other piece and that the square they were placed was not under attack by the opposition. Therefore, this also means the kings had to be at least two squares apart from each other.

APPENDIX C
Preset Starting Position

During the second testing phase agents were tested against a random player using a preset starting position. The starting position used does have a slight advantage to white when played by advanced players. Figure C.1 shows the starting position.

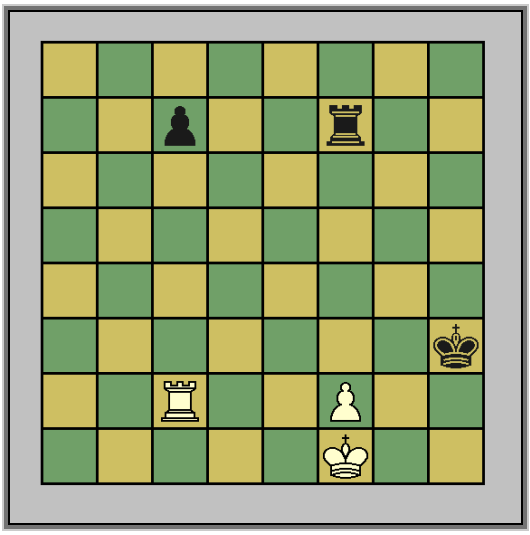
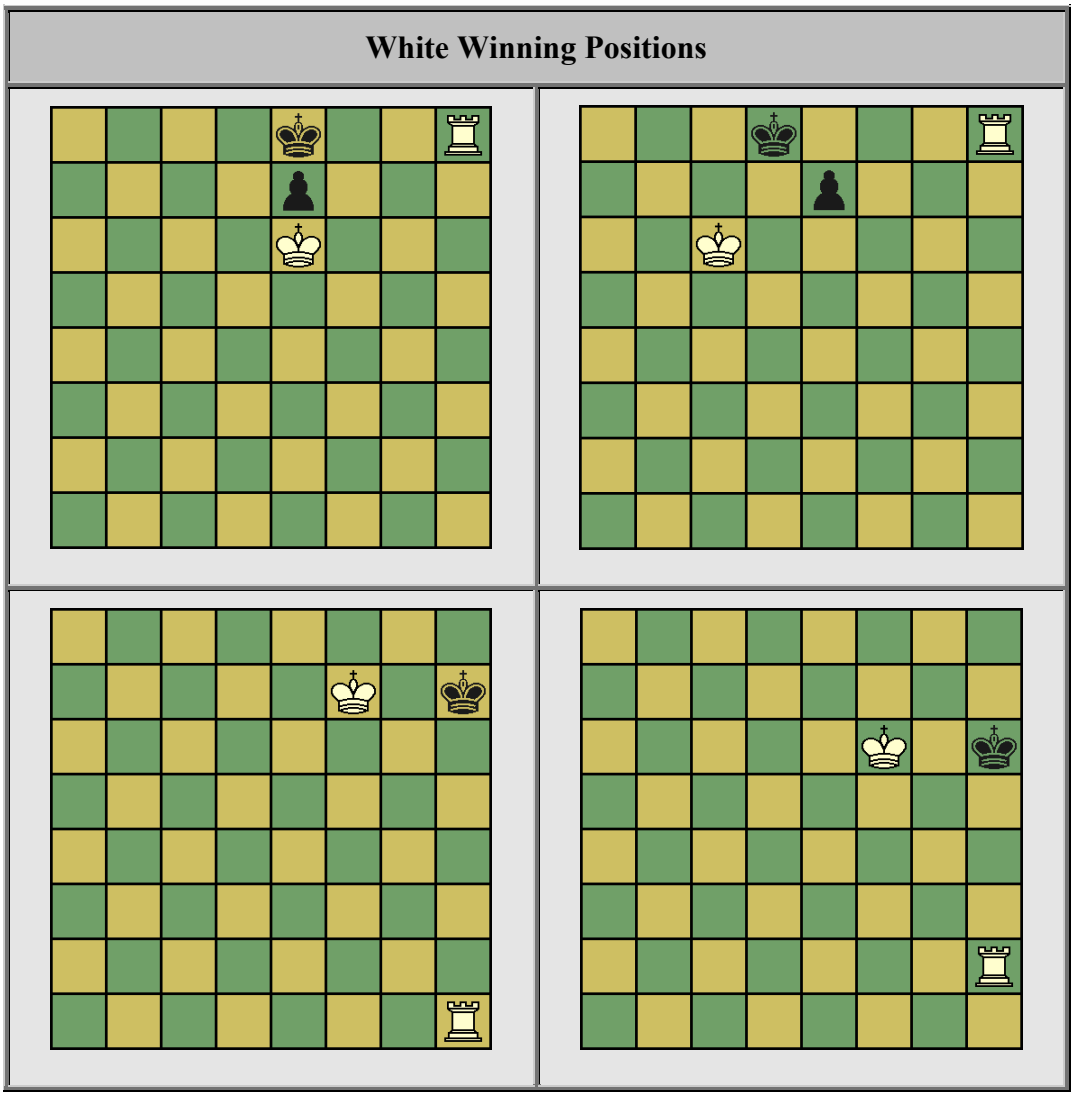


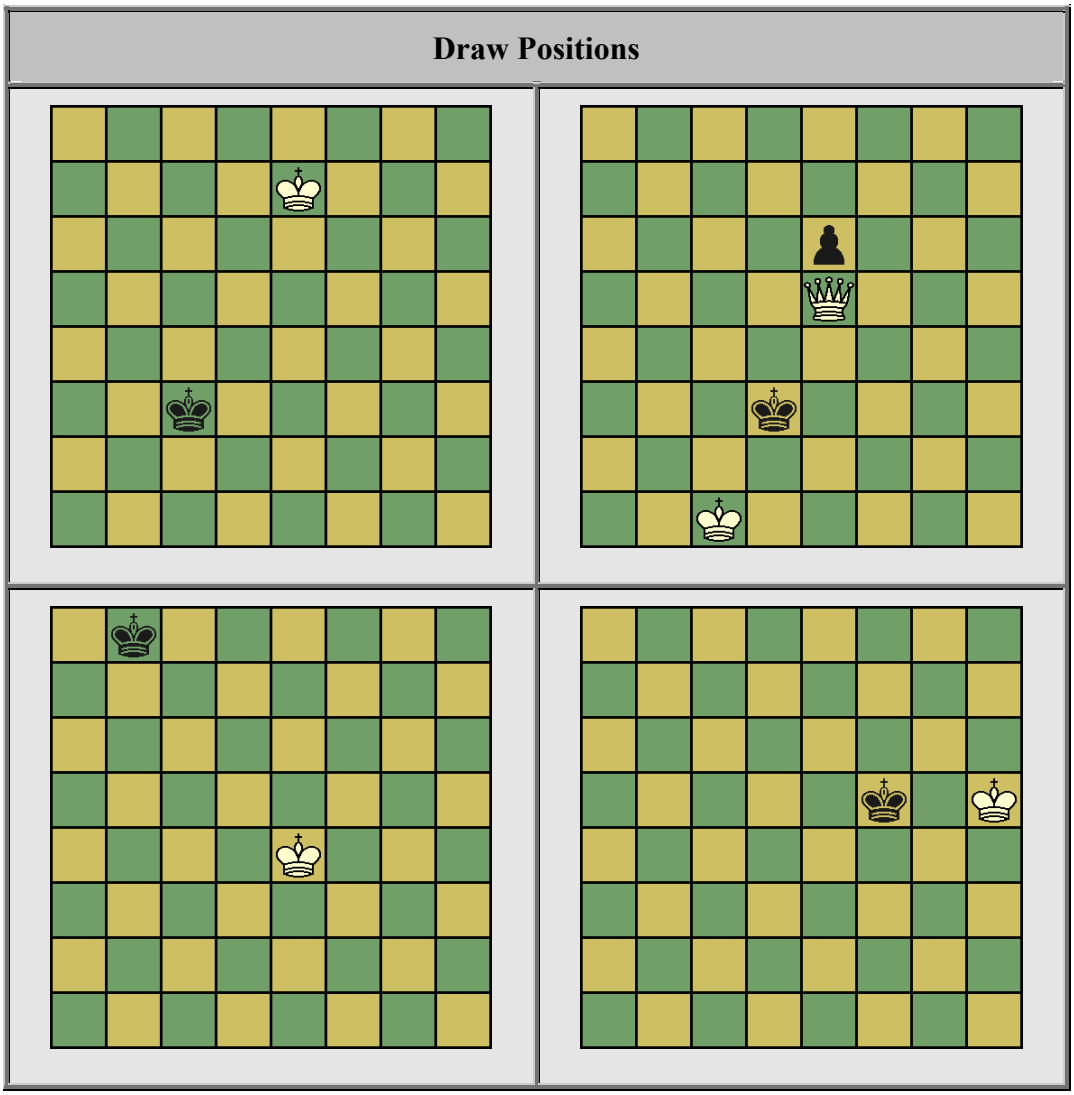
Figure C.1: Preset starting position.

APPENDIX D
Terminating Test States

The first phase of testing used twelve terminating positions to test whether the agents where able to differentiate between good and bad states. These twelve positions where broken up into three groups of four: 4 white winning, 4 draw and 4 black winning positions. Table D.1 gives these 12 board position.

White Winning Positions




Draw Positions

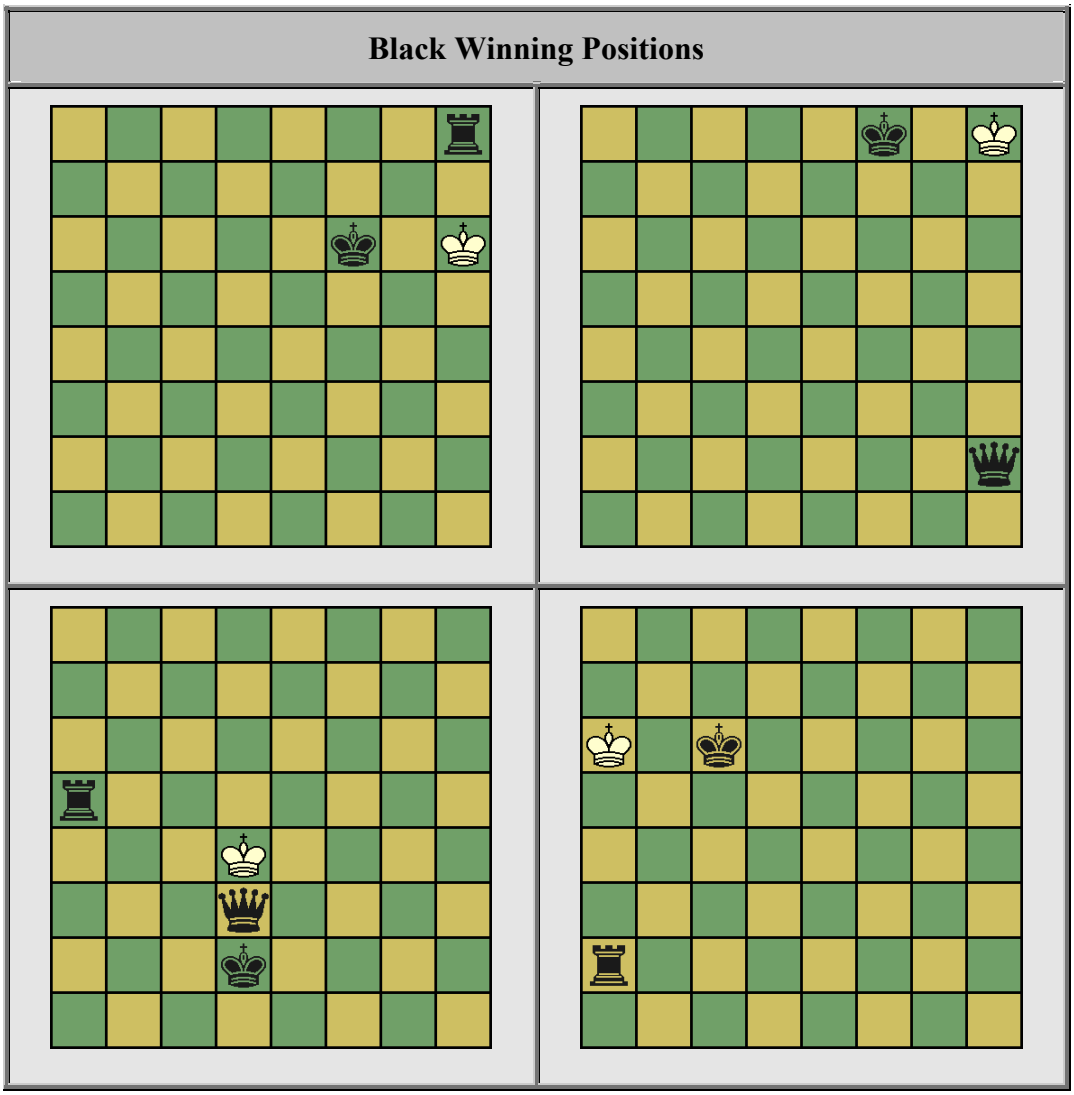


Table D.1: Terminating Positions used to test agents.

APPENDIX E
Terminating State Evaluations

Table D.1 contains a graph for each of the individual agent's terminating board position evaluations. Each graph has three vertical bars: white winning positions, draws and black winning positions. Each bar identifies the range from the minimum up to the maximum value that the example boards fed through the agent were awarded after training was completed.

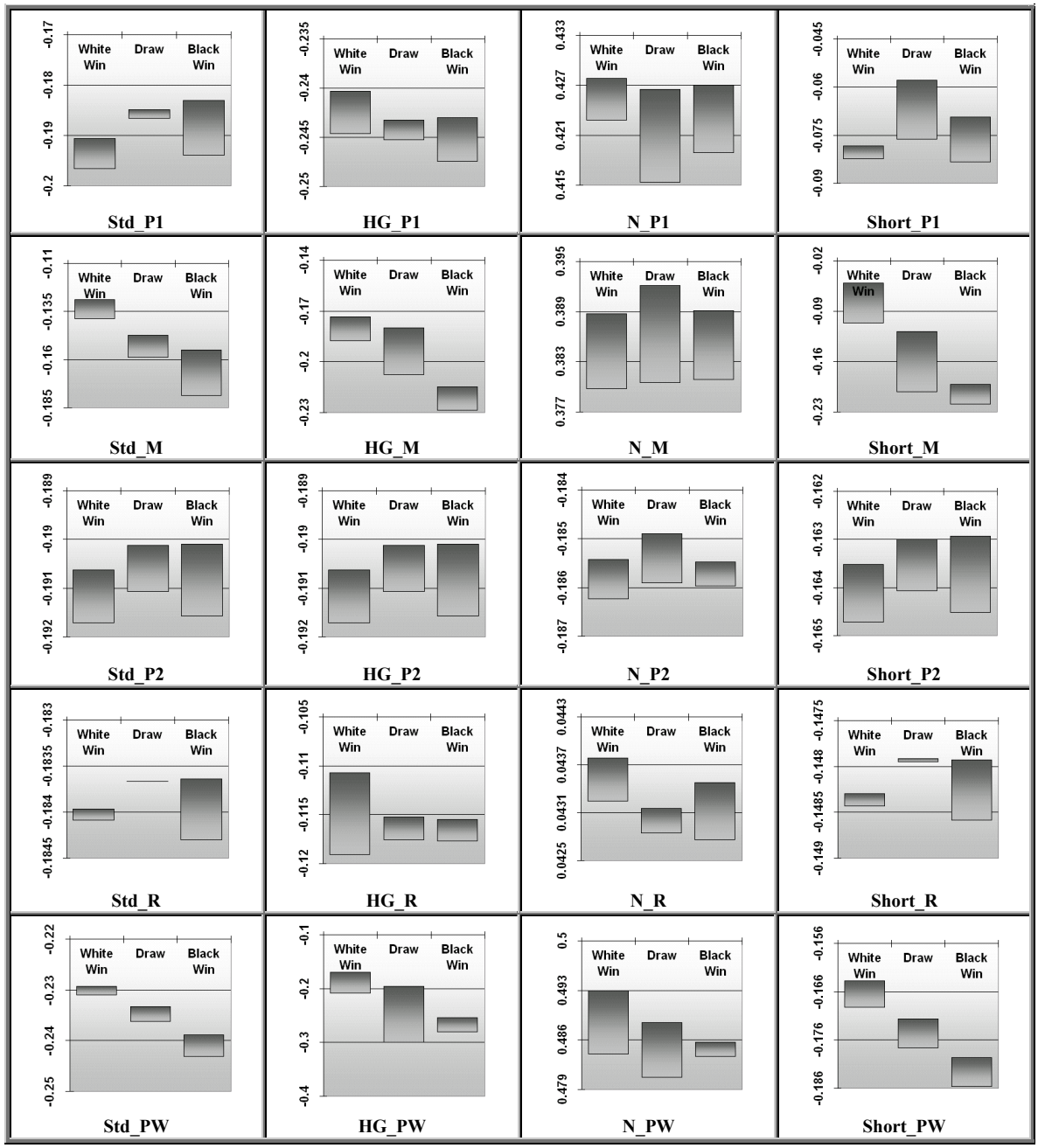
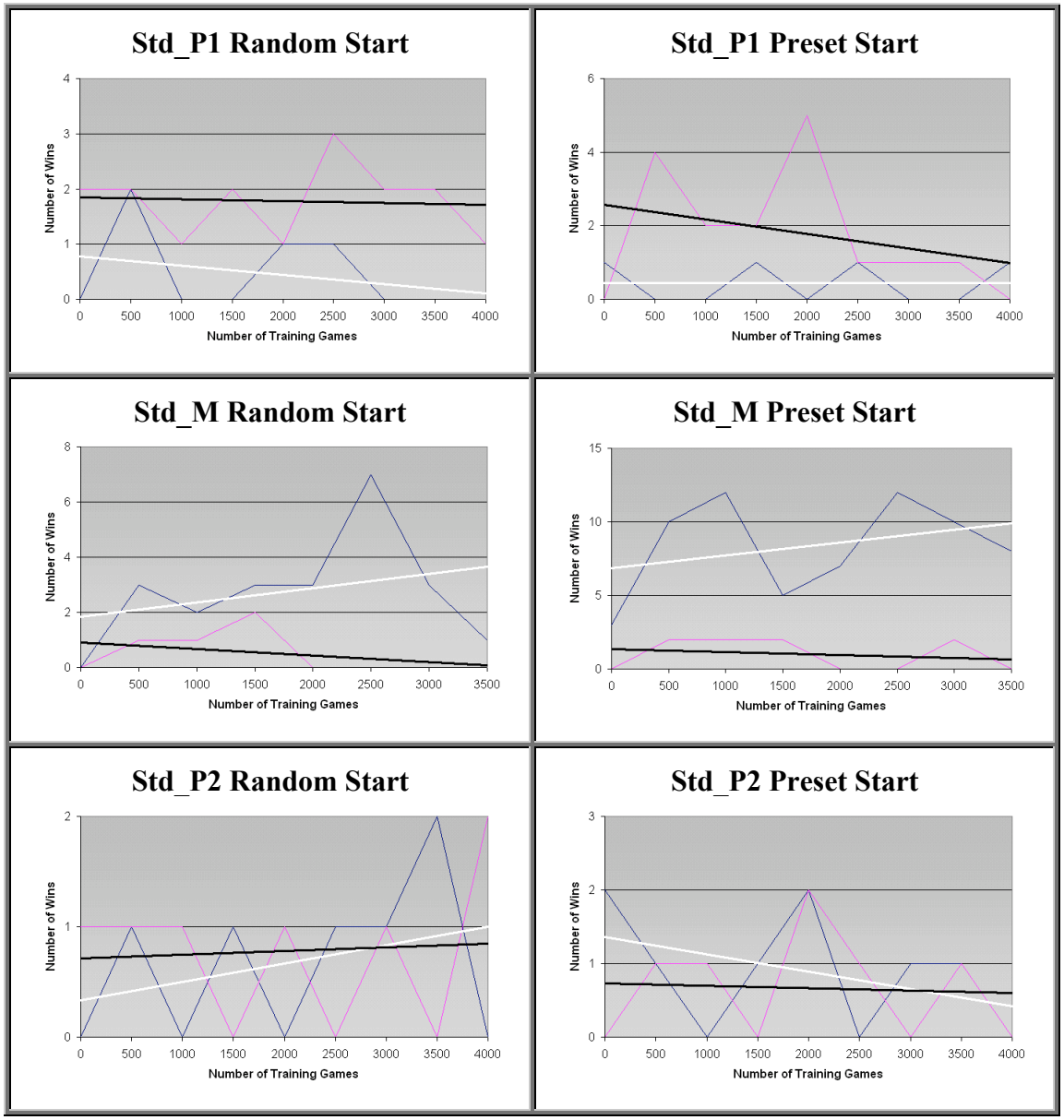


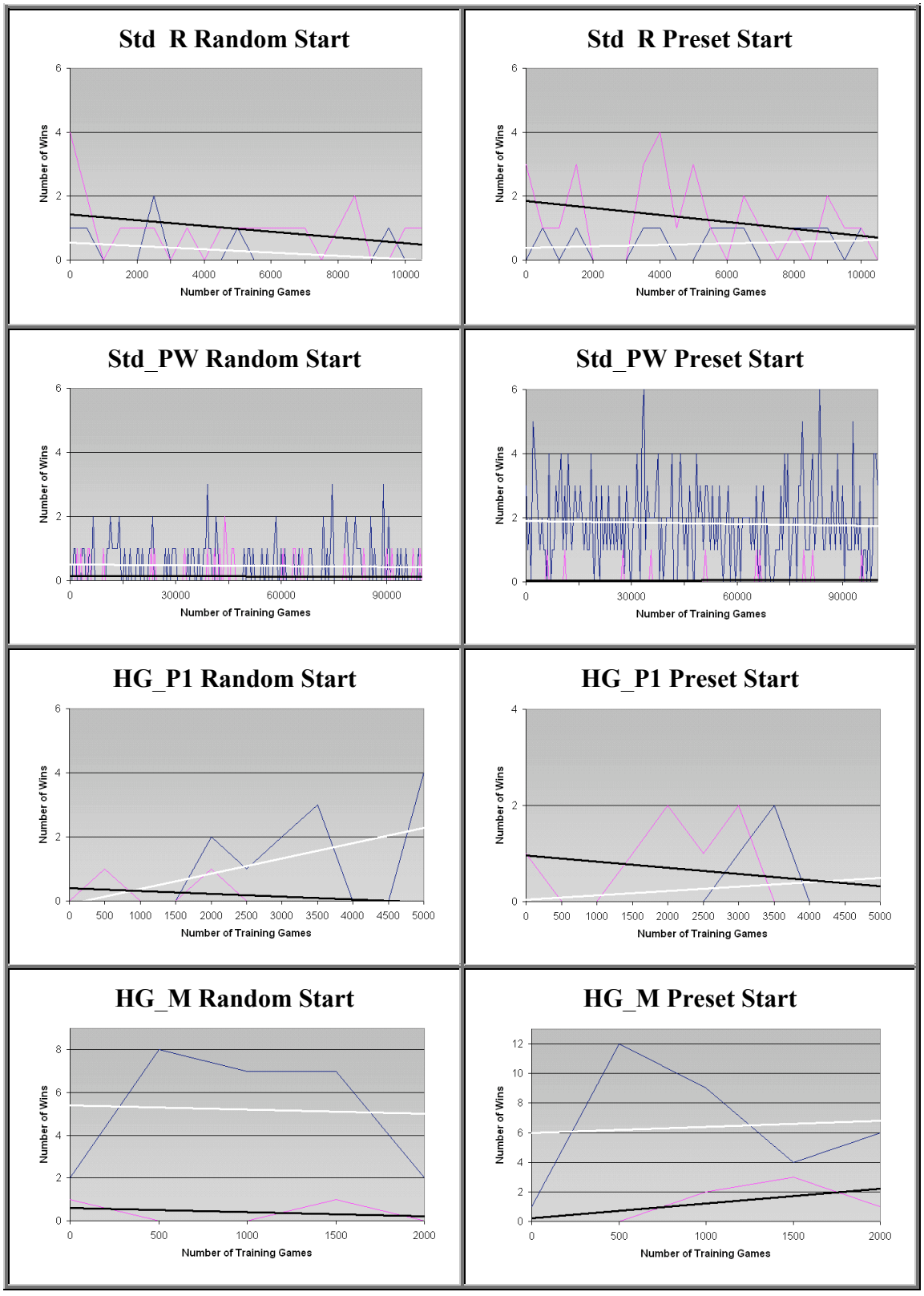
Table E.1: Graphs of Evaluations of Agents Terminating Positions.

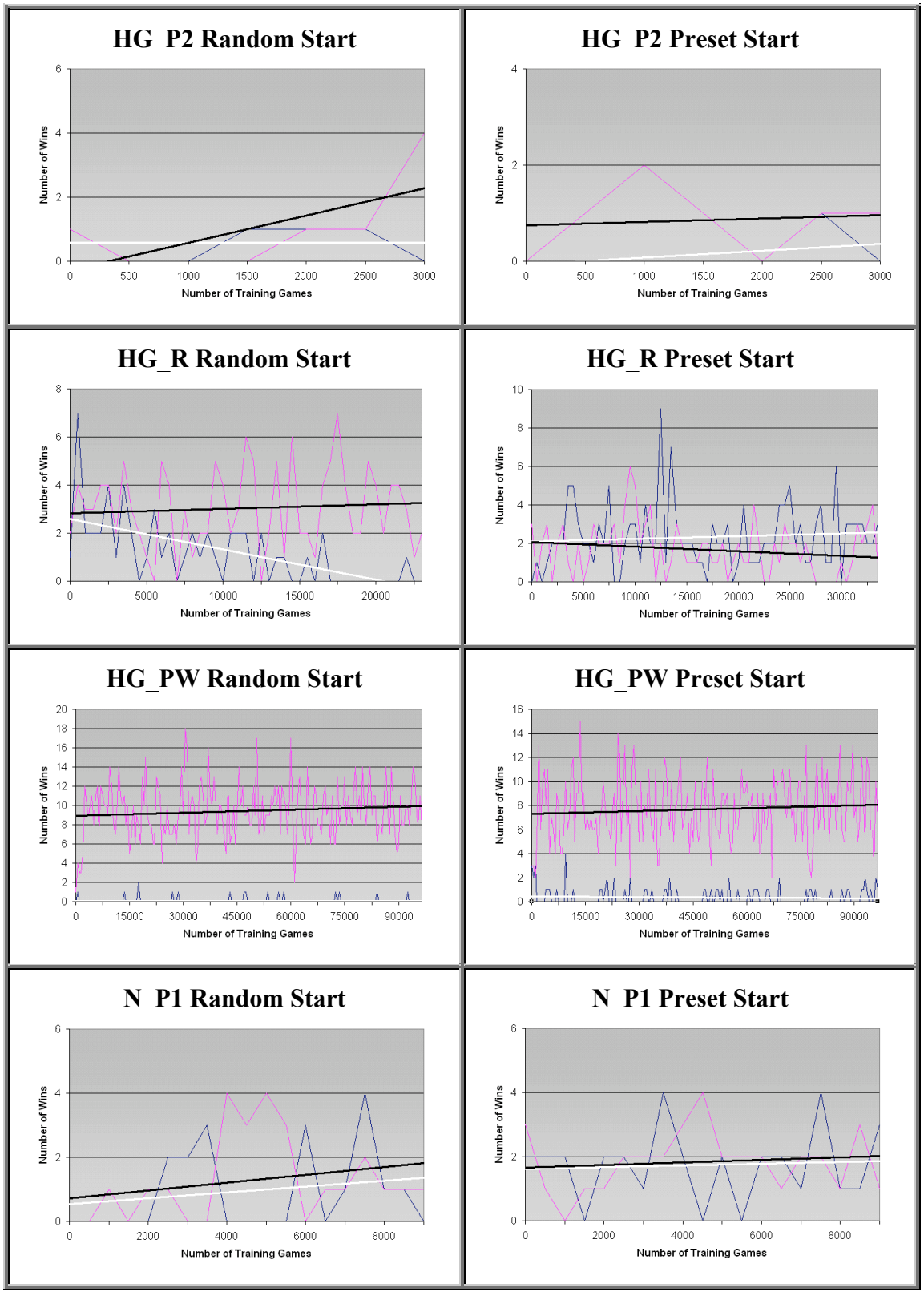
APPENDIX F

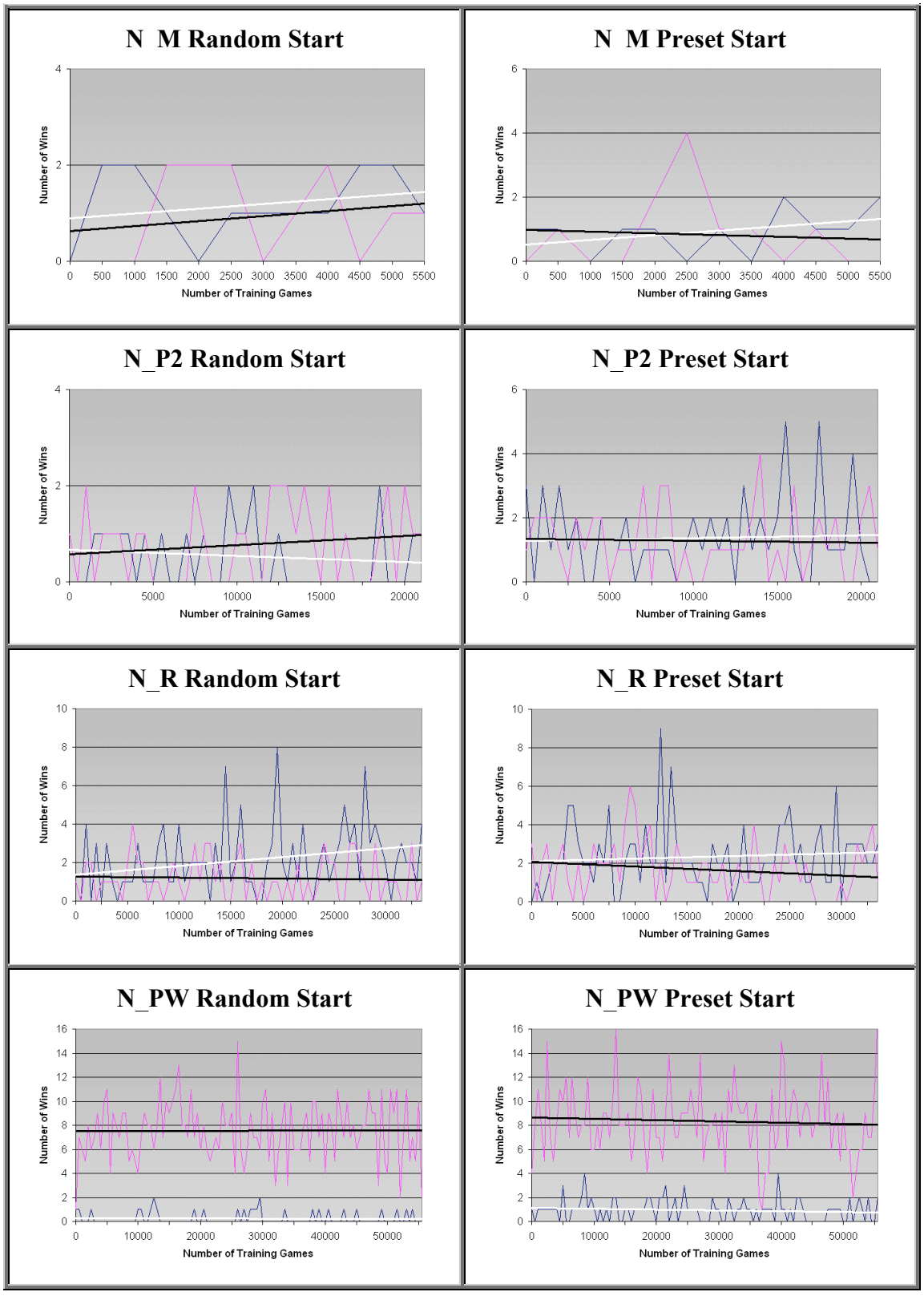
Agents vs Random Player

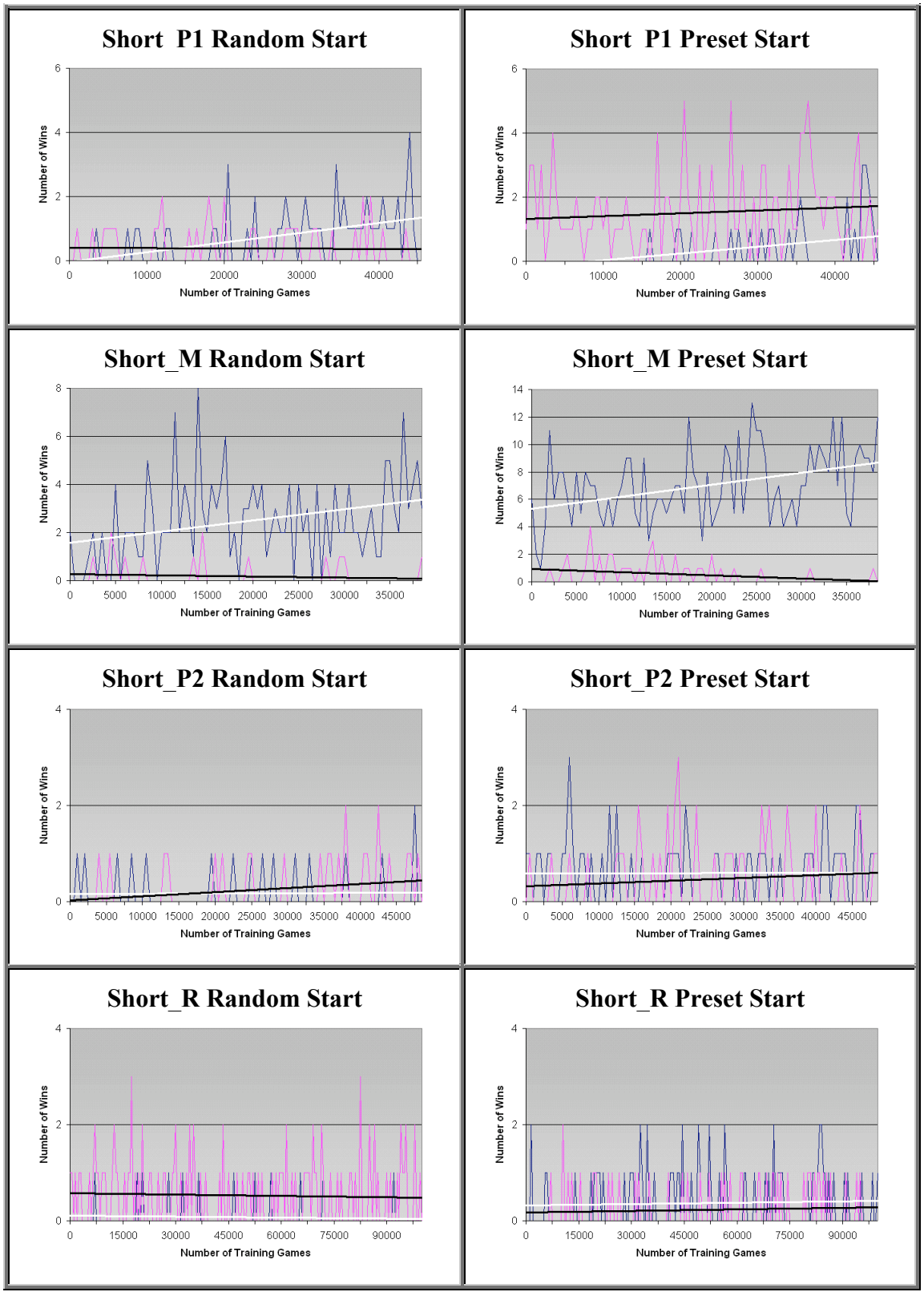
Table F.1 contains two graphs for each agent showing their performance against a random player. The first graph shows them playing with random starting positions and the second graph illustrates them playing with the preset starting position (Appendix B). Each graph has a white and black linear trend-line applied, showing the number of white and black wins respectively.




Std R Random Start
Number of Wins
Number of Training Games
Std R Preset Start
Number of Wins
Number of Training Games
Std_PW Random Start
Number of Wins
Number of Training Games
Std_PW Preset Start
Number of Wins
Number of Training Games
HG_P1 Random Start
Number of Wins
Number of Training Games
HG_P1 Preset Start
Number of Wins
Number of Training Games
HG_M Random Start
Number of Wins
Number of Training Games
HG_M Preset Start
Number of Wins
Number of Training Games


HG P2 Random Start
Number of Wins
Number of Training Games
HG P2 Preset Start
Number of Wins
Number of Training Games
HG_R Random Start
Number of Wins
Number of Training Games
HG_R Preset Start
Number of Wins
Number of Training Games
HG_PW Random Start
Number of Wins
Number of Training Games
HG_PW Preset Start
Number of Wins
Number of Training Games
N_P1 Random Start
Number of Wins
Number of Training Games
N_P1 Preset Start
Number of Wins
Number of Training Games


N M Random Start
Number of Wins
Number of Training Games
N M Preset Start
Number of Wins
Number of Training Games
N_P2 Random Start
Number of Wins
Number of Training Games
N_P2 Preset Start
Number of Wins
Number of Training Games
N_R Random Start
Number of Wins
Number of Training Games
N_R Preset Start
Number of Wins
Number of Training Games
N_PW Random Start
Number of Wins
Number of Training Games
N_PW Preset Start
Number of Wins
Number of Training Games


Short P1 Random Start
Number of Wins
Number of Training Games
Short P1 Preset Start
Number of Wins
Number of Training Games
Short_M Random Start
Number of Wins
Number of Training Games
Short_M Preset Start
Number of Wins
Number of Training Games
Short_P2 Random Start
Number of Wins
Number of Training Games
Short_P2 Preset Start
Number of Wins
Number of Training Games
Short_R Random Start
Number of Wins
Number of Training Games
Short_R Preset Start
Number of Wins
Number of Training Games

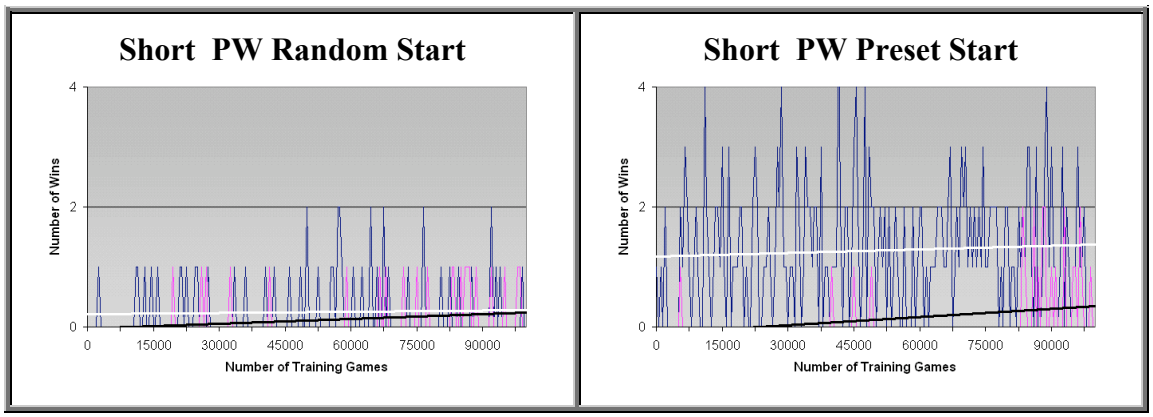


Table F.1: Graphs of Agents vs Random pl ayers using botyh random and preset starting positions.